Faculdade de Engenharia da Universidade do Porto

Defining a test automation system for mobile apps

Renato Rodrigues

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Dr. Rui Maranhão

Co-Supervisor: MEng. José Campos

July 10, 2014

Defining a test automation system for mobile apps

Renato Rodrigues

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Dr. Magalhães Cruz
External Examiner: Prof. Dr. José Maria Fernandes
Supervisor: Prof. Dr. Rui Maranhão

July 10, 2014

Abstract

Mobile devices like smartphones and tablets are everywhere. We are in an era where computers are being surpassed by these devices. Their portability and availability, compared to desktop and laptop computers, are seen as an advantage making their over-use more likely. Smartphones have become a central social, communication and information tool for many people. Statistics show that, in fact, people check their phones 150 times a day.

This wide range of devices with different operating systems results in thousands of different configurations. If a company wants to reach 95% of the active mobile market, it has to support more than 300 different devices. This immense diversity of devices is what is called by fragmentation, an ever-growing nightmare, not only for developers as well for end-users.

Such proliferation of mobile devices raises a big engineering problem. How to make sure an application will correctly work on most mobile devices in the market?

Although the web automation tools are very mature nowadays, the mobile ecosystem still has a long way to go. As the software running on mobile devices becomes more and more powerful and complex, the testing of these mobile applications poses great challenges for mobile application vendors. The purpose is very clear—create a continuous delivery solution with extensive tests, if possibly with tests on real devices. With functional testing in mind, one can test applications against the requirements specifications and hence validate them. But manual testing is out of question because it is expensive, slow and requires error-prone human workforce. Moreover, testing translates into repetitive tasks and such repetition is best-suited for computers. Mobile test automation frameworks arise as a viable solution to the problem in hands. These tools, if used on real devices and in an environment as close as possible to production, can give an elevated degree of confidence. Cloud-based testing services do exist and provide automated testing functionalities but cloud computing is yet a risk that most companies do not want to take. Security vulnerabilities, privacy concerns and possible outages and downtimes prevent companies from adopting these modern architectures.

This dissertation encompasses two goals: first, the optimization and refactoring of a continuous integration solution to achieve a continuous delivery solution, and, second, the design and implementation of a full system of automation testing that can be used to run tests across multiple physical mobile devices. A process that lives along with continuous integration practices and continuously drives tests on several physical mobile devices can greatly decrease the delivery cycle of a company.

The final solution was a robust deployment pipeline on an environment of continuous delivery with the inclusion of an process of automated acceptance tests being executed in a range of physical devices.

Resumo

Dispositivos móveis como *smartphones* e *tablets* estão em todo o lado. Vivemos numa altura em que os computadores estão a ser ultrapassados por estes dispositvos. A sua portabilidade e disponibilidade, em comparação com a dos computadores de secretária e portáteis, são vistos como uma vantagem tornando mais provável o seu uso prolongado. Os *smartphones* tornaram-se ferramentas de comunicação social e de informação para muitas pessoas. Estatísticas mostram que os utilizadores consultam o seu telefone mais de 150 vezes por dia.

Esta grande gama de dispositvos com diferentes sistemas operativos resulta em milhares de configurações diferentes. Se uma empresa pretender atingir 95% do mercado móvel activo, tem de suportar mais de 300 dispositivos diferentes. Este imensa diversidade de dispositivos resulta naquilo a que se chama de fragmentação, um pesadelo cada vez maior, não só para os programadores, bem como para os utilizadores finais.

Tamanha proliferação de dispositivos móveis levanta um grande problema de engenharia. Como certificar que uma aplicação irá funcionar corretamente na maioria dos dispositivos no mercado?

Embora as ferramentas de automatização *web* estejam bastantes maduras hoje em dia, o ecossistema móvel ainda tem um longo caminho a percorrer. À medida que o *software* executado nestes dispositivos móveis se torna mais e mais poderoso e complexo, o teste destas aplicações móveis apresenta grandes desafios para os fornecedores de aplicações. O objectivo é muito claro—criar uma solução de *continuous delivery* com testes extensos e, se possível, com testes em dispositivos físicos. Com testes funcionais em mente, pode-se testar aplicações contra especificações de requisitos e, assim, validá-las. Mas testes manuais estão fora de questão porque são caros, lentos e exigem mão-de-obra passível de erro humano. Além disso, testes traduzem-se em tarefas repetitivas e tal repetição é mais adequada para computadores. Ferramentas de automatização móvel surgem assim como uma solução viável para o problema em mãos. Estas ferramentas, se usadas em dispositvos reais e num ambiente o mais próximo possível do de produção, podem conferir um grau elevado de confiança. Existem serviços de testes na *cloud* que fornecem funcionalidades de automatização de testes mas *cloud computing* é ainda um risco que muitas empresas não querem tomar. Vunerabilidades de segurança, problemas de privacidade e a possibilidade de interrupções e falhas no serviço impedem as empresas de adotar estes modelos mais recentes.

Esta dissertação abrange dois objectivos: primeiro, a optimização de uma solução de integração contínua com vista a obter uma solução de *continuous delivery* e, segundo, o desenho e implementação de um sistema completo para a automatização de testes em múltiplos dispositivos móveis. Um processo que, usado em conjunto com práticas de integração contínua, executa testes em vários dispositivos móveis pode reduzir drasticamente os ciclos de produção.

A solução final é uma fila de produção (*deployment pipeline*) inteiramente automatizada num ambiente de *continuous delivery* com a inclusão de um processo com testes automatizados de aceitação executados num conjunto de dispositivos físicos.

Acknowledgments

This dissertation represents the final milestone of a laborious but marvelous journey with five years. Throughout these years, I have learned so much and I had the chance to know remarkable individuals. I would like to thank many people who have helped me through the completion of this dissertation.

First and foremost, I would like to express my gratitude to my supervisor, Prof. Dr. Rui Maranhão, whose expertise, knowledge and patience were essential for my success on writing this dissertation. I would like to thank my co-supervisor, José Campos, for the tireless support and patience. Your thoroughly feedback was extremely helpful.

A very special thanks go to Mauro Martins, my proponent and mentor at Blip, whose motivation and encouragement were essential to keep me pushing my limits. He provided me with direction, technical support and became more of a friend, than a proponent. I must also acknowledge my workmates: Luís Roma Pires, Tiago Correia, João Silva, Fábio Rocha, Pedro Baltarejo, Meik Schutz, Andreia Santos, Daniela Dias, Pedro Pimenta, Rui Pereira, Fábio Almeida, Carlos Santos, João David, Ricardo Mateus, Diana Barbosa, David Matellano, Sara Vilaça, Gonçalo Alvarez, Álvaro Monteiro, Ricardo Vieira, João Prudêncio and Rui Teixeira. I recognize the support of all you. Thank you for showing me such professionalism and patience with me. Besides them, a sincere thanks to everybody else at Blip that somehow I interacted with.

I must also thank the Group Buddies guys: Luís Zamith, Miguel Palhas and Bruno Azevedo. Your skills and knowledge were truly valuable. Together, we created a great solution.

I thank all my friends for all the support, patience and encouragement. There is no need to discriminate anyone. The culprits know well to whom I speak. A very special thanks goes to a friend who passed away. I am grateful for every single moment and, in part, you made me the man I am.

Last but not least, I would like to thank my family for all the support they provided me.

Renato Rodrigues

"Perfection is achieved not when there is nothing more to add,
but when there is nothing left to take away."

Antoine de Saint-Exupéry

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

List of Listings

List of Algorithms

Abbreviations

API	Application Programming Interface
BA	Business Analyst
BDD	Behavior Driven Development
CD	Continuous Delivery
CI	Continuous Integration
DM	Delivery Manager
DSL	Domain-specific language
GUI	Graphical user interface
HID	Human Interface Device
HTTP	Hypertext Transfer Protocol
Iaas	Infrastructure as Service
OC	Operating Context
OOS	Open-source software
OS	Operating system
PaaS	Platform as Service
PO	Product Owner
PSP	Potentially Shippable Product
QA	Quality Assurance
SaaS	Software as Service
SCM	Source Control Management
SUT	System Under Test
TaaS	Testing as Service
UDID	Unique Device Identifier
UI	User Interface
VCS	Version Control System
VM	Virtual Machine

Chapter 1

Introduction

This chapter intends to introduce this dissertation by discriminating its context and motivation, followed by the problem statement.

Section 1.1 contextualizes this dissertation and Section 1.2 explores the motivation of it. These two sections basically summarize why this dissertation emerged. Section 1.3 formally describes the problem this dissertation is encompassed to solve. The environment where all the research, experiments and implementation are going to be conducted is disclosed in Section 1.4. Lastly, Section 1.5 describes the structure of the rest of the dissertation.

1.1 Context and Framing

> *"It can take up to nine months to deploy an entertainment (mobile) application. But that is the duration of a cell phone in this market."*
>
> — *Craig Hayman, IBM*

Mobile devices like smartphones and tablets are everywhere[1]. We are in an era where computers are being surpassed by these devices [SS13]. Their portability and availability, compared to desktop and laptop computers, are seen as an advantage making their overuse more likely. "The mobile phone is no longer only a tool of communication but an indispensable instrument of an individual's social and work life.", as stated by [TTK09]. Smartphones have become a central social, communication and information tool for many people. In fact, people check their phones 150 times a day, according to Kleiner Perkins Caufield & Byers's annual Internet Trends report [MW13]. Smartphone ownership has

[1] http://www.smartplanet.com/blog/business-brains/milestone-more-smartphones-than-pcs-sold-in-2011/

never been so high. In May 2013, 91% of the world population owned a cell phone, of which 56% were a smartphone[2].

A immensity of devices exists to suit everybody tastes. Table 1.1 shows the different given classifications depending of the devices dimensions. Tablets have revolutionized the way people do simple tasks like read a book or edit a document—the same tasks that represented a challenge on smartphones.

Table 1.1: Classifications of mobile devices based on their dimensions[3]

Classification	**Dimension**
Smartphone	4.9″ and below
Phablet	5.0″ to 6.9″
Mini-Tablet	7.0″ to 9.6″
Tablet	9.7″ and above

This wide range of devices with different operating systems results in thousands of different configurations. According to a study conducted by Flurry[4] in May 2013, 150 is the number of devices to support if one wants to reach 80% of the active mobile market. And that number increases to more than 330 if trying to reach 90% of active mobile users. It is important to emphasize that, since the realization of this study, not only the number of devices has increased as well as the worldwide ownership percentage.

This phenomenon is called *mobile device fragmentation* where the term 'fragmentation' stands for the inability to develop an application against a reference operating context (OC) and achieve the intended behavior in all OCs suitable for the application[5]. An operating context for a mobile application is defined the hardware and software environment in the device as well as the target user, and other constraints imposed by various other stakeholders (such as the carrier). Fragmentation causes are shown in Table 1.2.

Such proliferation of mobile devices raises a big engineering problem that can be translated into two questions:

- How to make sure an application will correctly work on most mobile devices in the market?

- How to develop an application and test it thoroughly and automatically on so many devices with so many possible configurations?

[2] http://www.supermonitoring.com/blog/state-of-mobile-2013
[3] http://okjdiscoveries.wordpress.com/2013/10/02/how-to-choose-the-size-of-your-smartphone-or-tablet-the-orange-box/
[4] http://blog.flurry.com/bid/96368/There-s-An-App-Audience-for-That-But-It-s-Fragmented
[5] http://www.comp.nus.edu.sg/~damithch/df/device-fragmentation.htm

Table 1.2: Classifications of causes of mobile device fragmentation

Classification	Examples
Hardware diversity	screen parameters memory size processing power input mode additional hardware (e.g. camera or voice recorder) connectivity options
Platform diversity	differences in OS proprietary APIs variations in access to hardware differences in multimedia support
Implementation diversity	quirks of implementing standards
Feature variations	light version vs. full version
User-preference diversity	language accessibility requirements
Environmental diversity	diversity in the deployment infrastructure locale local standards

1.2 Motivation

> *"We can only make judgments about the suitability of a design based on real world usage. Testing is not an activity to be performed at the end of each project either; it is an integral part of the development process."*
>
> — *Paul Robert Lloyd*

As the software running on mobile devices becomes more and more powerful and complex, the testing of these mobile applications poses great challenges for mobile application vendors [BXX07].

The testing of mobile software differs significantly from the traditional software typically found on desktops and laptops. The unique features of mobile devices pose a number of significant challenges for testing and examining usability of mobile applications [ZA05].

Different screen sizes and resolutions

Mobile devices features different screen sizes and display resolutions. Applications should take account of the very different display capabilities as these physical constraints can affect the usability of mobile applications [JMMN+99]. Such significant range of different sizes means the same application has to be written for each screen

type—which with the non-stop proliferation of mobile devices is impracticable—or has to be able to work across the many different options available.

Data entry methods
Mobile devices feature different data entry methods. Hard keyboards are becoming history as the usage of soft keyboards is a reality of nowadays with the mass production of devices with touch capabilities. Different behaviors of use like having the device held in hand or put it on a table affects differently the effectiveness and efficiency in entering data. Another challenges emerge like multi-modal mobile applications that combine voice and touch inputs. "Blending multiple access channels provides new avenues of interaction to users, but it poses dramatic challenges to usability testing as well." [ZA05]

Limited processing capability and power
Computational power and memory capacity of mobile devices lag far behind desktops and desktops. Developers must pay attention to the code so that unwanted functions are not used and only the minimum amount of memory space is addressed. Code-efficient programming is imperative.

Connectivity
Mobile devices have the ability to connect to multiple services—in particular Internet, wireless carrier and GPS network—through device's Wi-Fi, Cellular and GPS module. Services availability may vary at different time and locations [Sea00]. Therefore, applications must be tested for the different forms of exposure a mobile device faces [DS14].

All these unique features lead to three key points that together rise as the motivation to create a solution in this dissertation:

Mobile development process poorly evolved Mobile development ecosystem is a relatively new field but it definitely came to stay. Being a new area, there is not a well-defined model assumed as the right one to be used. Different solutions and methodologies are emerging but it is up to each software company to adopt its method and adapt it to the company culture.

Multiple mobile devices with multiple configurations By the end of 2013, there was more mobile devices on Earth than people [Cis13]. According to study conducted by Super Monitoring [6], their mobile growth statistics for 2013 clearly shows an unstoppable growth of the mobile adoption. Some interesting statistics:

- 91% of all people on earth have a mobile phone
- 56% of people own a smart phone

[6] http://www.supermonitoring.com/blog/state-of-mobile-2013

- 80% of time on mobile is spent in applications
- The average consumer actively uses 6.5 applications throughout a 30-day period.

With this proliferation of mobile devices, it is very important to test applications on different devices, different operative systems and different configurations.

Emulators and simulators do not find all faults Software emulators and simulators can be useful but in the end they can only do that: simulate the experience. For example, iOS push notifications do not work in the simulator because such feature depends on a unique device token which simulators do not have. Emulators are great for rapid smoke-tests but one can not rely on them. There might be issues that are not spot during the test phase—assuming only emulators are used—that will only appear when the application is used on an actual mobile device.

For instance, Game Oven Studios, a game company, delayed the launch of a product because they found at the last minute their application had different behaviors depending of the device they used to run it [7]'. They tested the same compass application on seven different Android devices. Yet, all compasses indicated that North was somewhere else. This erroneous behavior had nothing to do with electromagnetic fields confusing the compass but with the diversity of hardware inside the used devices. When they expanded their list of test devices, they ended up founding that:

1. some devices had 'broken' gyroscopes that did not work on all axis;
2. that some devices were faking gyroscopes by mixing and matching the accelerometer data with compass data;
3. that some devices did not have a gyroscope at all.

This real example demonstrates how important is to test a mobile application on different devices.

Apple itself advises the developers to test on a variety of devices. "Rigorously test your application on a variety of devices and iOS versions. Because different kinds of devices and iOS releases have different capabilities, it is not sufficient to test your application on a device provisioned for development or the simulator. iOS Simulator does not run all threads that run on devices, and launching applications on devices through Xcode disables some of the watchdog timers. At a minimum, test the application on all devices you support and have available. In addition, keep prior versions of iOS installed on devices for compatibility testing. If you do not support certain devices or iOS versions, indicate this in the project target settings in Xcode."[8]

[7] http://gameovenstudios.com/bounden-on-android-delayed/

[8] https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/TestingYouriOSApp/TestingYouriOSApp.html

With this three points in mind, the purpose is very clear—*drive tests on real devices* in order to test extensively the functionality of a mobile application on a continuous basis. Chapters 4 and 7 discriminate why such approach of testing only on devices is not practicable. With a fully automated hand-to-hand testing framework, every new piece of code is guaranteed to meet the minimum quality requirements and, therefore, the company may rest assured that their products, after every update, are a Potentially Shippable Product[9] (PSP).

1.3 Problem Statement

> *"Computers are designed to do simple repetitive tasks. The second you have humans doing repetitive tasks, all the computers get together late at night and laugh at you…"*
>
> *— Neal Ford*

Although the web automation tools are very mature nowadays, the mobile ecosystem still has a long way to go. What is missing is a fully automated tool—or rather, a toolset—with the ability to run tests on different devices in a parallel way, that through automated and self-testing builds running on multiple real devices, certify that an application meets its functional requirements.

This dissertation started by only encompassing the design and implementation of a system for automated testing that could be used to run tests across multiple iOS devices as part of the continuous integration processes with an iOS project at Blip as test case. Quickly, the scope of the dissertation was extended due to its intrinsic relation with the overall process of software development. A new objective emerged involving not only the optimization of the continuous integration processes but the achievement of a continuous delivery solution as well. The concepts of continuous integration and continuous delivery are described in the next chapter.

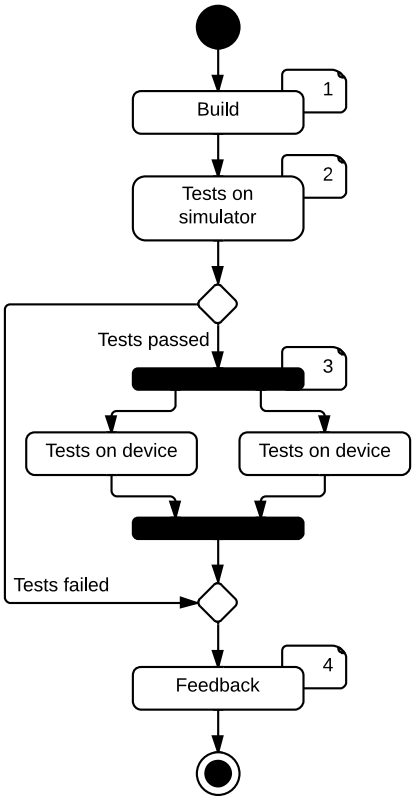
Both the implementation of the automated platform for testing and the conversion to a continuous delivery environment bring designs issues that requires decisions regarding the technologies and approaches to be used.

Implementing a system capable of running tests across a wide range of devices with different configurations is a delicate task. Its architecture must be flexible enough to allow developers to manage the connected devices—disconnecting existing devices or connecting new ones. For a completely automated system, the use of a continuous integration solution will be imperative. A central server will first test the applications in a simulator and only if the tests succeed the applications are sent to the connected devices. The testing results of every mobile device must be collected back to the server in order to give feedback to developers and testers.

[9] A Potentially Shippable Product is a product good enough to be shipped. It is up to the Product Owner the decision to ship it or not.

The very foundation of this system is centered on having a test farm, a collection of one or more servers, which has been set up to extensively test mobile applications remotely. Under a cross-platform testing scenario, each server [10] can test the latest version of an application on a different device. One of the difficulties of cross-platform development is that a programmer may unintentionally introduce an error that causes the software to stop functioning on a different device from the one they are using—even more problematic when a programmer only uses a simulator. By using a cross-platform test farm, such errors can be identified and fixed. By testing on multiple devices, a greater level of confidence can be achieved.

This UI automation system should be as automated as possible so the developers and testers just have to commit their code to a repository and receive back, as soon the tests are done, feedback (including reports and build results). To be able to call it a fully automated system, the final product must ensure a pipeline of automated actions as shown in Figure 1.1.



Step 1: Build the application with the most recent code—whenever someone commits changes to the respective repository.

Step 2: Run a suite of automated tests on an appropriate mobile device simulator. If the tests fail, skip the tests on the devices.

Step 3: Install the application to the connected devices in the network and run again the suite of tests.

Step 4: Collect the results and broadcast them to the proper individuals.

Figure 1.1: UI automation system main actions

If all the tests pass then it means that not only the new code did not created conflicts but the application as a whole remains functional as well. It also means the new features

[10] It is also possible to have a cross-platform behavior with a single server but it requires a setup in a parallel fashion.

work properly, assuming that for every new feature, new tests were written.

On continuous delivery, the concept itself is straightforward but its implications affect a lot the organization. In short, it represents a philosophy and a commitment to ensure that a project code is always in a release-ready state. It is a step forward from continuous integration, the process of automatically building and testing software on a regular basis. A solution of this type requires automated steps of building and testing capable of giving enough confidence to the team about the product quality. Figure 1.2 shows a high-level point of view of the solution. The UI automation system previously described is naturally included as a step of this solution—represented as *Run automated UI tests* on the diagram—since it not only represents a big confidence indicator but the most important one as well.

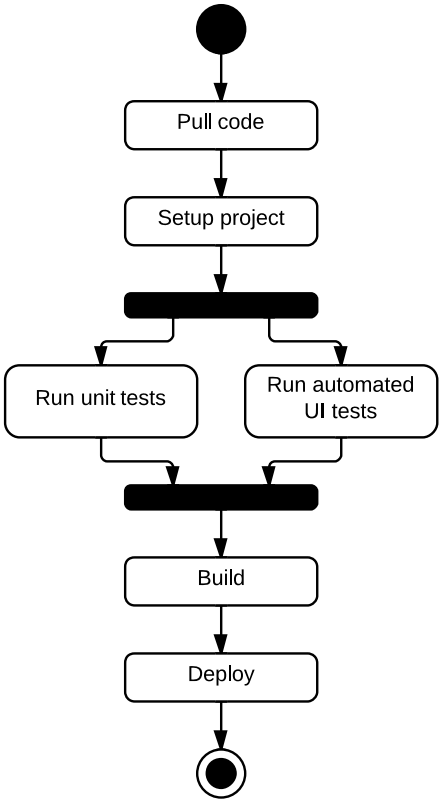


Figure 1.2: Continuous delivery strategy

Several questions arise with the problem stated. Adequate approaches and technologies must be studied in order to determine the path to follow.

- What is the proper approach to take to have an automated pipeline?
- How to improve the level of confidence on the software under test?
- How to connect multiple mobile devices in the network?
- How to collect back the testing results from the devices?

- Logical tests are covered but how to enforce usability tests that only real people can lead?

1.4 Case Study

All the research and findings required to write this dissertation are going to be done in an enterprise environment at Blip, a Portuguese company established in Oporto whose public statement is to have a passion for developing high performance Web products using agile methodologies to get the best out of every sprint [11].

All projects of the company follow an agile approach which promotes radical changes when compared to how software development companies traditionally work [TKHD06]. Software testing not only is no exception as it is the most affected field. Formal iterative life cycle methods allow substantial periods of time between test execution periods for each iteration. Agile methodologies like Scrum are less formal and faster moving. The main characteristic of these methodologies are the short, fast-paced iterations called sprints. Testing strategies that include an automatic element have proven particularly sensitive to this challenge. Once again, this dissertation aims to create a solution that fully automates all the acceptance tests in the sprints.

1.5 Dissertation Structure

Besides this introduction, this dissertation has seven other chapters. Chapter 2 introduces essential concepts required for the rest of the dissertation. In Chapter 3, the state of the art of the topics contemplated by this dissertation is presented in two main sections—mobile test automation frameworks and cloud-based testing. Chapter 4 describes the solution implemented through four sections—starting point, a general overview of where the solution started from; deployment pipeline, automated acceptance tests and feedback are the other three sections that describe in detail the solution. All the experiments conducted are listed and described in Chapter 5. Chapter 6 presents an abstract view of the implemented solution that can be reproduced by anyone. Chapter 7 is an important chapter that analyses all the decisions made and measures the pros and cons of the tools and approaches used. Finally, Chapter 8 summarizes the work done by way of reflection and presents ideas of future work to further improve the developed solution.

[11] http://www.blip.pt/about

Chapter 2

Background

In this chapter, some fundamental concepts to better understand the rest of the dissertation are introduced.

Section 2.1 gives a brief explanation about software testing and a more detailed description on functional testing, an important concept as one of the goals of dissertation is to develop a solution of automated acceptance tests. Section 2.2 introduces a tool called Cucumber used to write functional tests. Sections 2.3 and 2.4 describe the concepts involved with continuous integration and continuous delivery.

2.1 Functional Testing

> *"Testing is a cross-functional activity that involves the whole team and should be done continuously from the beginning of the project."*
>
> *— Jez Humble and David Farley*

The number of test scenarios for a mobile application increases with each new feature, new view or bug discovered. So many scenarios represent a serious problem to test them all manually and so emerges the need for an automated solution. With this in mind, follows the question of what kind of tests should one automate.

The essence of software testing is to determine a set of tests cases for the item to be tested. A complete test case will contain a test case identifier, a brief statement of purpose (e.g., a business rule), a description of preconditions, the actual test case inputs, the expected outputs, a description of expected postconditions, and an execution history [Jor13]. Two fundamental approaches are used to identify test cases: structural and functional testing—also called code-based and specification-based, respectively, which are more descriptive names.

Functional testing got such name because any program can be considered to be a function that maps values from its input domain to values in its output range. This notion is commonly used in engineering, when systems are considered to be black boxes. This led to another synonymous term—black box testing, in which the content (implementation) of the black box is not known. Code-based testing is the other fundamental approach to test case identification. To contrast it with black box testing, it is sometimes called white box testing. The essential difference is that the implementation (of the black box) is known and used to identify test cases. For instance, unit tests which fit in the white box testing category play an crucial role in software testing but are focused on specific features or components, therefore they are not the most suitable to test how all parts of an application work together.

Functional testing is a quality assurance process and a type of black box testing that focus on the overall behavior of the application, as opposed to unit tests that aim to only test an isolated component. It usually describes what the system does, therefore the internal program structure is rarely considered. In other words, these tests check if the application does what it is supposed to do, ensuring that the behavior of the system adheres to the requirements specification and are often performed based on use-cases written in natural language, usually a domain language—a language with terms related to the application domain. Another acceptance criteria is the visual appearance of the application, such as brand issues and design guidelines. Being the last set of tests performed before putting an application into production, they should be as realistic as possible, meaning that they should be executed on an environment very identical to the production as a way to increase the confidence that the application works.

2.2 Cucumber

Cucumber[1] is a tool that executes plain-text functional descriptions as automated tests. It lets software development teams describe how software should behave in plain text. The language that Cucumber understands is called Gherkin.

While Cucumber can be thought of as a "testing" tool, the intent of the tool is to support Behavior Driven Development (BDD). This means that the "tests" (plain text feature descriptions with scenarios) are typically written before anything else and verified by business analysts, domain experts or other non technical stakeholders. The production code is then written outside-in, to make the stories pass. Behavior driven development is an "outside-in" methodology.[2] It starts at the outside by identifying business outcomes, and then drills down into the feature set that will achieve those outcomes. Each feature is captured as a "story", which defines the scope of the feature along with its acceptance criteria.

[1] http://cukes.info/
[2] http://dannorth.net/whats-in-a-story/

A story has to be a description of a requirement and its business benefit, and a set of criteria by which everyone agree that it is *done*. An BDD story looks like the template in Listing 2.1.

```
Title (one line describing the story)

Narrative:
As a [role]
I want [feature]
So that [benefit]

Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given [context]
  And [some more context...]
 When [event]
 Then [outcome]
  And [another outcome...]

Scenario 2: ...
```

Listing 2.1: BDD story template

Cucumber follows this approach but instead of calling it a story, it calls a *feature*. Every `.feature` file conventionally consists of a single feature.[3] A line starting with the keyword *Feature* followed by free indented text starts a feature. A feature usually contains a list of scenarios.

Every scenario consists of a list of steps, which must start with one of the keywords *Given*, *When*, *Then*, *But* or *And*. Listing 2.2 shows an example of a Cucumber feature file.

```
Feature: Serve coffee
  Coffee should not be served until paid for
  Coffee should not be served until the button has been pressed
  If there is no coffee left then money should be refunded

Scenario: Buy last coffee
  Given there are 1 coffees left in the machine
  And I have deposited 1$
  When I press the coffee button
  Then I should be served a coffee
```

Listing 2.2: Cucumber feature example

[3] https://github.com/cucumber/cucumber/wiki/Feature-Introduction

For each step, Cucumber will look for a matching *step definition*. A step definition is written in Ruby. Each step definition consists of a keyword, a string or regular expression, and a block. Step definitions can also take parameters if you use regular expressions. Listing 2.3 shows a step definition with a regular expression.

```
1 # features/step_definitions/coffee_steps.rb
2
3 Given /there are (\d+) coffees left in the machine/ do |n|
4   @machine = Machine.new(n.to_i)
5 end
```

Listing 2.3: Step definition example

2.3 Continuous Integration

> *"In the term 'continuous integration',* integration *refers to assembly of software parts and* continuous *to the absence of time-constraints."*
>
> — *Jesper Holck and Niels Jørgensen*

When speaking of automation, an indispensable concept arises known as *continuous integration*.

Continuous integration (CI) describes a set of software engineering practices that speed up the delivery of software by decreasing integration times. Its aim is to prevent integration problems. By eliminating repetitive tasks, theses practices seek to minimize "bad builds" and improve product quality. Such practices to work must be executed continuously, that is, several times a day. Figure 2.1 shows a common approach of continuous integration solution [FF06].

CI is becoming a mainstream technique for software development as stated by Martin Fowler [FF06], a software engineer, specializing in agile software development methodologies, who helped create the Manifesto for Agile Software Development[4]. He defines the practices of CI as follows:

Maintain a single source repository All artifacts required to build the project should be placed in a repository through the use of a revision control system.

Automate the build It must be possible to build the system with a single command. A script must take care of all the required process to put the system together.

Make your build self-testing A suite of automated tests must run after the code is built because checking if the program runs is not enough. The failure of a test should cause the build to fail.

[4] http://agilemanifesto.org/

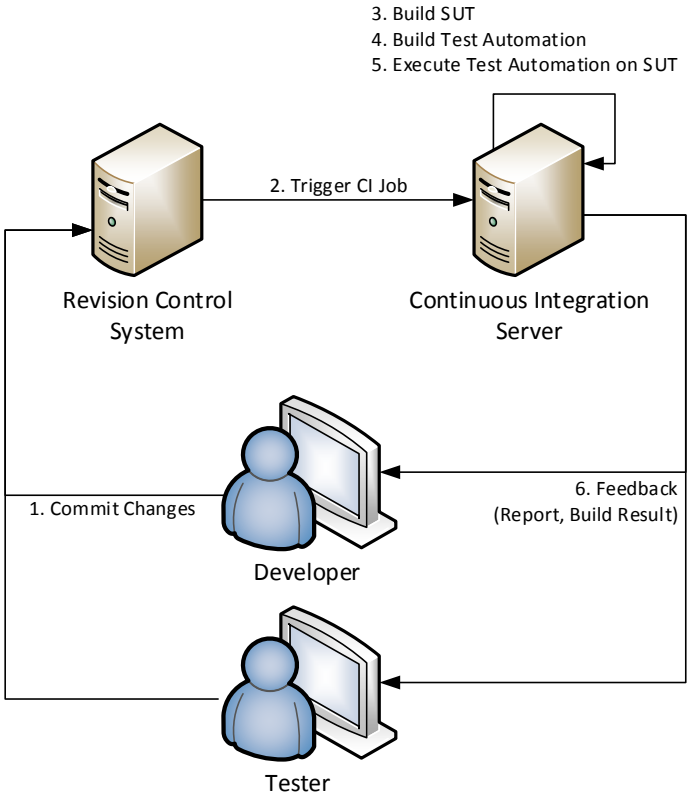


Figure 2.1: Automated continuous integration solution

Everyone commits to the mainline every day By committing regularly, developers quickly find out if there is conflicts. "The key to fixing problems quickly is finding them quickly."

Every commit should build the mainline on an integration machine Using manual builds or a continuous integration server, regular builds should happen on an integration machine. The commit is only considered to be done if these integration builds succeed.

Keep the build fast To have rapid feedback, the build needs to complete rapidly. The sooner a problem is identified, the sooner a developer can fix it.

Test in a clone of the production environment The goal is to duplicate the production environment as much as possible because testing in a different environment might result in behaviors than do not happen in production.

Make it easy for anyone to get the latest deliverables Anyone involved with the project—stakeholders and testers, for instance—should be able to get the latest executable and be able to run it.

Everyone can see what's happening Everyone should be able to see, at any moment, the state of the system and what have changed.

Automate deployment It is important to have scripts that allow to deploy the application into different environments such into a test server or even into production (an emerging practice known as continuous deployment) after additional automation to prevent defects or regressions.

2.4 Continuous Delivery

> *"Testing and deployment can be a difficult and time-consuming process in complex environments comprising application servers, messaging infrastructure and interfaces to external systems. We have seen deployments take several days, even in cases where teams have used automated builds to ensure their"*
> — *Jez Humble, Chris Read and Dan North*

Continuous delivery (CD) is the logical step forward from continuous integration. If tests are run constantly, and one trusts the tests to provide a guarantee of quality, then it becomes possible to release the respective software at any point in time. It represents a philosophy and a commitment to ensuring that one code is always in a release-ready state[5]. When working on a CD environment, developers used to a long cycle time may need to change their mindset because any code commit may be released to customers at any point.

Continuous integration is an enormous step forward in productivity and quality for most projects that adopt it. It ensures that teams working together to create large and complex systems can do so with a higher level of confidence and control than is achievable without it [HF10]. CI is primarily focused on asserting that the code compiles successfully and passes a body of unit and acceptance tests. However, CI is not enough.

CI mainly focuses on development teams. The output of the CI system normally forms the input to the manual testing process and thence to the rest of the release process. The main drawback in releasing software is the progress through testing and operations. Typical scenarios include testers having to wait for *good builds* or discovering, towards the end of the development process, that the application's architecture will not support the system's nonfunctional requirements. This waste of time leads to software that is undeployable because it has taken so long to get it into a production-like environment, and buggy because the feedback cycle between the development team and the testing team is so long. This paves way for possible improvements to the way software is delivered such as writing production-ready software and running CI on production-like systems. However, while these practices will certainly improve the process, they still do not give an insight into where the bottlenecks are in the delivery process or how to optimize them.

One solution to these problems is to automate fully the build, testing and deployment process, which should be done in the early stages of the project so it can saves developers

[5] http://blogs.atlassian.com/2014/04/practical-continuous-deployment/

time and, more importantly, it also helps detect problems with deployment early in the development cycle, where fixing the problem is cheaper. Furthermore, automating the entire deployment process embodies a key agile practice, making the code—in this case, the deployment scripts—the documentation []. As a result, the build and deployment scripts capture the deployment and environment testing process, and can be leveraged to give rapid feedback not just on the integration of the modules of the project's code, but also on the problems integrating with the environment and external dependencies [HRN06].

Over the course of many projects, [HF10] identified much in common between the continuous integration systems they have built. Their solution was to turn the overall process in an autonomous *deployment pipeline* through the automation of the build, deploy, test and release processes. They took it to a point where deploying their applications would only require a simple click of a button. An environment with such characteristics results in a powerful feedback loop—since it is so simple to deploy an application to testing environments, the team gets rapid feedback on both the code and the deployment process. "Since the deployment process (whether to a development machine or for final release) is automated, it gets run and therefore tested regularly, lowering the risk of a release and transferring knowledge of the deployment process to the development team." This deployment pipeline revolves around a set of validations through which a piece of software must pass on its way to release.

There are two key points in order to achieve rapid feedback. In the first place, the pipeline must start with the fastest steps because the sooner a step in the pipeline fails, the quicker everyone will be able to trace the problem. Secondly, the pipeline must start with *showstoppers*, that is to stay, essential steps that must always pass in order to get a *good* build. As one moves along the pipeline, it goes from showstoppers to not necessarily ones. For instance, building and unit testing must always complete successfully—if something fails it is known the product is broken, however on the manual testing phase—e.g., through exploratory testing—one can find a bug not severe enough that prevents the release of the product. The environments should also become more production-like as this will increase the confidence in the build's product readiness, mainly in the acceptance test and manual testing stages.

One way to better understand the deployment pipeline and how changes move through it is to visualize it as a sequence diagram—as first described in [HF10], as show in Figure 2.2.

The input of the pipeline is a particular revision in version control. Every change creates a build that will pass through a sequence of tests to acquire its viability as a production release. "This process of a sequence of test stages, each evaluating the build from a different perspective, is begun with every commit to the version control system, in the same way as the initiation of a continuous integration process." [HF10] As the build passes each test of its fitness, confidence in it increases. Therefore, the resources that one is willing to expend on it increase, which means that the environments the build passes through

become progressively more production-like. The objective is to eliminate unfit release candidates as early in the process as we can and get feedback on the root cause of failure to the team as rapidly as possible. To this end, any build that fails a stage in the process will not generally be promoted to the next. These trade-offs are shown in Figure 2.3.

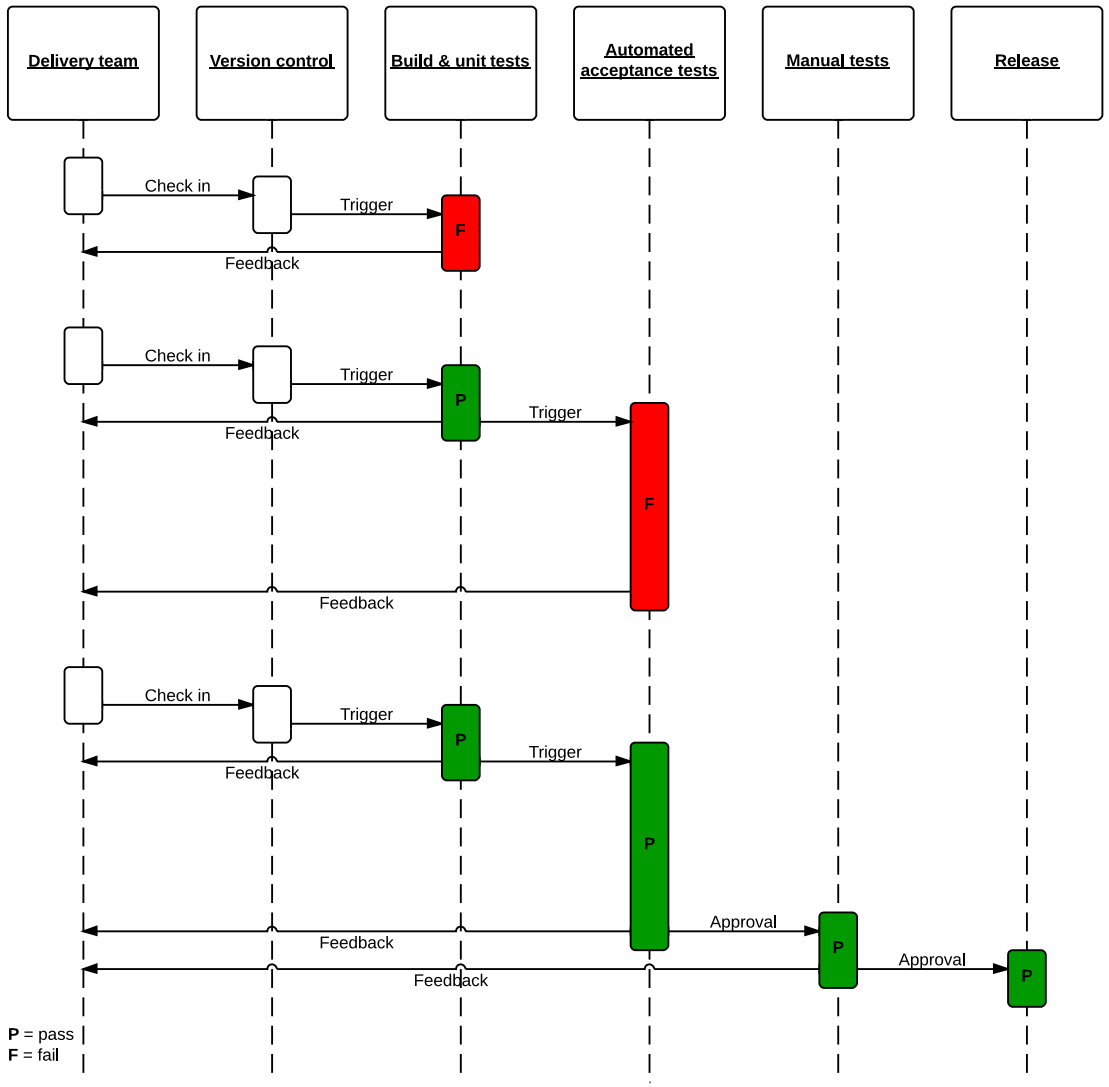


Figure 2.2: Processes sequence through the deployment pipeline

A brief explanation of each phase follows:

- The building and unit testing phase—or, as some literatures name it, the *commit stage*—asserts that the system works at the technical level. There are three objectives to be met: the syntax of the source code must be valid so a executable can be created, unit tests must pass and, lastly, certain quality criteria such as test coverage and other technology-specific metrics must be fulfilled.

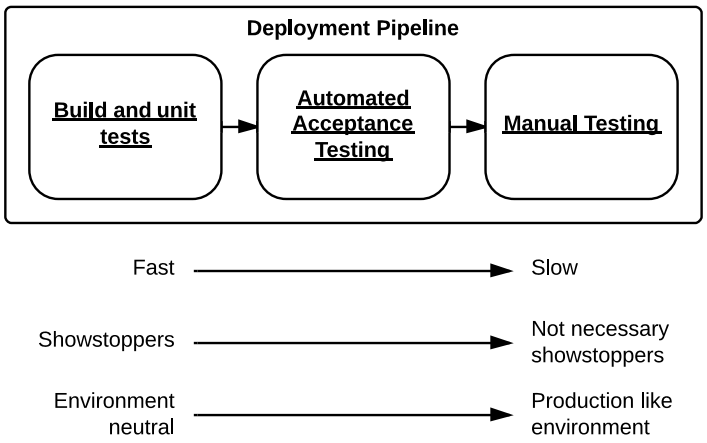


Figure 2.3: A basic deployment pipeline and its trade-offs

- *Automated acceptance test stages* assert that the system works at the functional and nonfunctional level by verifying if its behavior meets the needs of its users and the specifications of the customer.
- *Manual test stages* verify if the system is usable and fulfills its requirements. It is an imperative phase because it let detect any defects not caught by automated tests.
- *Release stage* delivers the system to users, either as packaged software or by deploying it into a production or staging environment[6].

These stages, and any additional ones that may be required to model the process for delivering software, are referred to as *deployment pipeline*. It is essentially an automated software delivery process but it not implies having no human interaction with the system through the release process; rather, "it ensures that error-prone and complex steps are automated, reliable and repeatable in execution" [HF10].

[6] A staging environment is a testing environment identical to the production environment.

Chapter 3

Related Work

This chapter describes the state of the art about existing work relevant to the context of the problem.

Section 3.1 introduces the concept behind mobile test automation and lists the major frameworks as well as its pros and cons. Section 3.2 describes cloud-based services used for the testing of mobile applications. After an overview of cloud computing, the major services for testing automation are listed and compared. In the end, a set of advantages and disadvantages are listed, after witch some conclusions on the subject are made. Section 3.3 points the final verdict regarding the framework to be used on the final solution.

3.1 Mobile Test Automation Frameworks

Mobile testing is a real challenge when it comes to automation because of variety of the platforms and systems. Developers seek to create and use automated tools to test their applications on different devices. A job that performed manually would be too slow and expensive.

3.1.1 Test Automation Principles

All user interface (UI) test automation tools rely on three basic mechanisms to define and execute tests[1]:

UI element locators allow the tester to locate UI elements to act upon

Human Interface Device (HID) event generation allows the tester to simulate actions—keystrokes, mouse clicks, touch events and similar HID events—on a UI element in order to drive the system under test.

[1] http://testautomation.applitools.com/post/62062545011/can-image-based-functional-test-automation-tools

UI data extraction allows the tester to read data from a UI element in order to validate the correct behavior of the SUT.

3.1.2 Mobile Automation Technologies

Mobile automation technologies fall into two techniques: instrumented and non-instrumented. *Instrumentation* refers to the process of inserting of inserting code into an application. An *instrumentation framework* is a software system that allows an entity to insert instrumentation at specific points in a program [HLHG13].

In technologies based in instrumented techniques, tests are compiled with the application and consequently installed and launched with the application. Hence, the source code is required (which may need to be modified) and only one application can be executed at a time. While instrumentation techniques uses text-based features, non-instrumentation ones rely on external image based features allowing external and hardware interactions. Table 3.1 summarizes advantages of both techniques.

Table 3.1: Advantages of instrumentation and non-instrumentation[2]

Technique	Advantages
Instrumentation	Elements can be accessed Debugging ease Test verification ease Reduce tools dependencies Support for: • installing, launching and killing application • test execution on device • code coverage
Non-instrumentation	Device platform agnostic Test code reuse Test language and test harness autonomy Support for: • multi-application testing • custom UI elements • database/server API assertions • use of external libraries

Section 3.1.3 presents a list containing the most used frameworks for mobile automation where only 2 out of the 11 presented frameworks use non-instrumented techniques. Such low popularity is consequence of a newest generation of tools based around the idea of driving applications using keywords and actions rather than click-by-click scripting[3,4].

[2] http://www.slideshare.net/eingong/2012-java-onecon3648

[3] http://testautomation.applitools.com/post/62062545011/can-image-based-functional-test-automation-tools

[4] http://testobsessed.com/2007/02/functional-test-tools-the-next-generation/

The tools in this case are image-based and while on the one hand it allows a person with no knowledge in programming create tests, on the other hand, these same tests are much more susceptible to changes and, hence, more difficult to maintain. Because these tools rely on UI element images for identification, changes in the system under test can broke the tests, forcing the images of all affected UI elements to be recaptured and adjusted manually by the tester. For the purpose of functional test automation, which focuses on testing the functionality of applications rather than their visual appearance, text-based tools—also called object-based— are by far the preferred tools of choice.

3.1.3 Major Frameworks

A list of the major automated frameworks follows.

Appium[5] supports native Android and iOS applications and mobile web. On Android, Appium executes the tests as either uiautomator (if API level 16 or greater) or Selendroid, and on iOS the tests are executed via UIAutomation.

Calabash[6] is a cross-platform mobile test automation technology to write and execute automated acceptance tests which currently supports iOS and Android native applications and web applications. It consists of libraries that enable test-code to programmatically interact with native and hybrid applications. The interaction consists of a number of end-user actions where an action can be one of the follows:

Gestures Touches or gestures (e.g. tap, swipe or rotate)

Assertions For example: *There should be a "Login" button*

Screenshots A screen dump of the current view on the device

It is a behavior driven framework. Tests are described in Cucumber and then converted to Robotium or Frank in run time. It runs on physical devices as well as simulators/emulators.

eggPlant[7] is a image-based testing framework for native iOS, Android, Blackberry and Windows applications.

Espresso[8] is the latest Android test automation framework from Google and works as a layer on top of Android Instrumentation Framework.

Frank[9] is an open source UI testing tool based on UISpec for native iOS applications. It requires modification of the application under test source code, inclusion of static

[5] http://appium.io/
[6] http://calaba.sh/
[7] http://www.testplant.com/eggplant/
[8] http://code.google.com/p/android-test-kit/wiki/Espresso
[9] http://www.testingwithfrank.com/

resources and source files. Because Frank's touch emulation is derived from UISpec, it only supports tapping.

MonkeyTalk[10] automates functional interactive tests of iOS and Android applications. Its tests work on native, mobile and hybrid applications, either in real devices or simulators.

Robotium[11] is an Android test automation framework that fully supports native, hybrid and web testing. Robotium can run tests in parallel mode against several devices and supports gestures (e.g. scrolling, touching or dragging). It also supports screenshots. Main disadvantage of Robotium is that it cannot work with different applications, only offering control of the one under test.

Sikuli[12] uses image recognition to identify and control UI components. Sikuli is not a mobile automation solution but automation can be achievable through the use of emulators or VNC on real devices since it is a image-based framework.

UIAutomation[13] is a framework for iOS native applications with gesture support that can be run both on device and simulator. Apple's UIAutomation framework is quite advanced in supporting various gestures and it is also simple to set up since it works without modification of the application but is has major down sides that make it hard to use in the long term: impossible to extend, small changes in UI cause massive changes in tests and cannot be integrated in a CI environment.

uiautomator[14] is Google's test framework for testing native Android applications but it only works on Android API level 16 and greater.

UISpec[15] is a behavior driven development framework that provides full automated testing solution that drives the actual iPhone UI.

Table 3.2 summarizes the supported platforms by each framework and whether the tests can be executed or not in emulators and real devices.

3.1.4 Conclusions

eggPlant and Sikuli are two image-based frameworks which, as discussed in Section 3.1.2, have into more disadvantages than advantages, mainly with functional testing. UIAutomation, despite of being a good framework, cannot be integrated in a CI environment.

[10] https://www.cloudmonkeymobile.com/monkeytalk
[11] https://code.google.com/p/robotium/
[12] http://www.sikuli.org/
[13] https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html
[14] http://developer.android.com/tools/help/uiautomator/index.html
[15] https://code.google.com/p/uispec/

Table 3.2: Comparison of automated frameworks

Framework	Supported platforms			Tests done on	
	iOS	**Android**	**Mobile Web**	**Emulator**	**Device**
Instrumented					
Appium	✓	✓	✓	✓	✓
Calabash	✓	✓	✓	✓	✓
Espresso		✓	✓	✓	
Frank	✓			✓	✓
MonkeyTalk	✓	✓	✓	✓	✓
Robotium		✓	✓	✓	✓
UIAutomation	✓			✓	✓
uiautomator		✓		✓	
UISpec	✓			✓	✓
Non-instrumented					
eggPlant	✓	✓	✓	✓	
Sikuli	✓	✓		✓	✓

The solution sought in this dissertation aims to rely on such environment, what eliminates UIAutomation from the acceptable tools. Espresso, uiautomator and Robotium only support either iOS or Android applications. Although Blip currently only wants an automated solution for iOS testing, to find at this stage a framework capable of testing Android applications too is an better option in the long term.

This selection leaves four frameworks—Appium, Calabash, Frank and MonkeyTalk. All these tools are very powerful. Frank only supports iOS, while Calabash is very similar to it but has also support for Android. Appium is not so mature as MonkeyTalk and Calabash. Finally, between these two, there is no much difference but Calabash supports Cucumber which let one to write tests with BDD style. This is a great advantage since it lets development teams describe how software should behave in plain text that serves as documentation and automated tests.

3.2 Cloud-based Testing

Cloud-based testing revolves around the concept of *cloud computing*.

3.2.1 Cloud Computing

"Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those ser-

vices" [AFG+10]. It is a relatively recent term, which basically defines a new paradigm for service delivery in every aspect of computing that enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. A cluster of computer hardware and software—called *Cloud*—that offer services to the general public (probably in a pay-as–you-go manner) is called a *Public Cloud*. The service being sold is *Utility Computing* and is offered as utility much like electricity or water where one only pays per use. The Service and Deployment models of cloud are shown in Figure 3.1.

Service models	**Deployment models**
Infrastructure as Service (Iaas)	Public Cloud
Platform as Service (PaaS)	Private Cloud
Software as Service (SaaS)	Community Cloud
Testing as Service (TaaS)	Hybrid Cloud

Figure 3.1: Cloud computing models [BC13]

3.2.2 Cloud Testing of Mobile Systems

Testing a service over cloud is a highly topical issue [IAS12, Vil12, YTC+10] that includes testing services for mobile development [CR12] known as Mobile Testing-as-a-Service (or simply Mobile TaaS). Some developers are using TaaS systems based on cloud services that provide the ability to run tests on a variety of remote mobile devices.

"Mobile Testing-as-a-Service provides on-demand testing services for mobile applications and SaaS to support software validation and quality engineering processes by leveraging a cloud-based scalable mobile testing environment to assure pre-defined given service-level-agreements (SLAs)."[16] Through a pay-as-you-test model, cost-sharing and cost-reduction in resources can be achieved.

To meet the challenges exposed in Chapter 1, many of TaaS systems serve as a *cloud of devices* that provides a wide range of remote mobile devices for manual or automated testing [SV13]. Such services allow developers and testers to the applications on various devices and networks remotely over the Internet.

3.2.3 Major Services

A variety of cloud testing services exist to facilitate mobile applications testing. A list of the major services follows.

Keynote DeviceAnywhere[17] provides a service where the devices are connected by hard-wiring. All operation is done trough a test framework that requires software installation.

[16] http://2013.icse-conferences.org/documents/publicity/AST-WS-Gao-Tsai-slides.pdf
[17] http://www.keynotedeviceanywhere.com/

Perfecto Mobile[18] provides a cloud-based service for many mobile devices that makes use of a USB data power to connect to devices. The service also provides an optional high-resolution camera. It is a web-based solution so no installation is required.

SOASTA TouchTest Mobile Labs[19] is another cloud-based service that allows developers and testers test their applications on real devices.

Xamarin Test Cloud[20] is a very recent but promising service that supports testing of iOS and Android applications (native or hybrid). It supports Behavior Driven Development (BDD) as it uses Calabash.

SOASTA TouchTest Mobile Labs and Xamarin Test Cloud are little mature compared to the other two. TouchTest uses an *record and replay* approach which means the developer has to capture all the gestures to create the tests. The service then replays those gestures in different devices. This approach is very susceptible to changes in the UI which may happen in two different cases: when the UI changes depending on the resolution or when the code itself is updated reflecting changes in the overall UI. Xamarin Test Cloud, still in beta, looks very promising. This service provides features much like the other services but it has the advantage of supporting Calabash. Table 3.3 compares various factors of DeviceAnywhere and Perfecto Mobile, the two most mature services[21,22].

Cloud-based architecture gives teams the ability to access different devices anywhere in the world, eliminating the need to have a physical device on hand to test. Mobile Test-as-a-Service is an emerging model with great advantages but it also comes with many downsides.

3.2.4 Advantages

Cloud computing offers numerous advantages. The obvious one is that there is no need to support the infrastructure running the services. A list of the most important advantages follows.

Abundance of resources

Such services can provide access to real mobile devices (different models, possibly including latest releases) on different carriers.

Rental of devices

Developers and testers can rent specific devices on which they want to test their applications and the option to swap devices is available as well.

[18] http://www.perfectomobile.com/
[19] http://www.soasta.com/products/touchtest-mobile-labs/
[20] http://xamarin.com/test-cloud
[21] http://mobiletestingtools.wordpress.com/2012/12/12/perfecto-mobile-vs-keynote-deviceanywhere/
[22] http://www.automatedtestinginstitute.com/home/index.php?option=com_k2&view=item&id=2654:comparison-of-mobile-automation-capabilities&Itemid=231

Web-based interfaces

Most of these tools do not require software installation, allowing ease of use. Only an Internet connection and a browser are required.

Ease of integration

A cloud system can be up and running in a very short period, making quick deployment a key benefit. Usually, mobile cloud-based testing solutions offers integration options with CI practices and frameworks. Some services provide integration with testing systems like HP Unified Functional Testing (HP UFT)[23] (for instance, Perfecto Mobile has HP UFT support).

Parallel testing

Tests can be run on several devices in parallel which greatly reduce the time needed to run all the tests in every device.

Debugging functionalities

Automated test execution is recorded to video to investigate failures as well as device logs to help troubleshooting.

3.2.5 Disadvantages

As made clear from the above, cloud computing is a tool that offers enormous benefits to its adopters. However, being a tool, it also comes with its set of problems and inefficiencies. A list of the most significant ones follows.

Subscription model

Subscription model may result in high costs. Pay-as-you-test model is not cost effective for mobile automation in large-scale. Because of the licensing cost involved, this architecture is more justifiable for pilot or very short term projects.

Dependency and vendor lock-in

One of the major disadvantages of cloud computing architectures—Mobile TaaS without exception—is the implicit dependency on the provider, a phenomenon called *lock-in*. It is often difficult, and sometimes impossible, to migrate from a provider to another because APIs for cloud computing are still essentially [AFG+10]. Thus, some organizations are avoiding from adopting cloud computing.

Limited control and flexibility

Since the applications and services run on remote, third party virtual environments, companies and users have limited control over the function and execution of the hardware and software. Moreover, latency may occurs depending how far the client

[23] http://www8.hp.com/us/en/software-solutions/unified-functional-testing-automated-testing/index.html

location is from the device location. Automation cannot be used outside most of the frameworks and certain types of testing are not possible (e.g. Bluetooth or interruptions), for instance, in applications that require peripheral devices.

Availability of service

Outage and downtime are a possibility of cloud service providers. "If people went to Google for search and it was not available, they would think the Internet was down. Users expect similar availability from [cloud computing] services..." [AFG+10] Table 3.4 shows recorded outages for Amazon Simple Storage (S3), AppEngine and Gmail in 2008, and explanations for the outages, which proves that even the best infrastructures are no exception to availability problems.

Table 3.4: Outages in AWS, AppEngine and Gmail

Service and outage	Duration
S3 outage: authentication service overload leading to unavailability	2 hours
S3 outage: single bit error leading to gossip protocol blowup	6–8 hours
AppEngine partial outage: programming error	5 hours
Gmail: site unavailable due to outage in contacts system	1.5 hours

Security and privacy

Security is a big concern when it comes to cloud computing. "By leveraging a remote cloud based infrastructure, a company essentially gives away private data and information, things that might be sensitive and confidential."[24] Another concern is that many nations have laws requiring SaaS providers to keep customer data and copyrighted material within national boundaries. "Current cloud computing infrastructure solutions are not capable of enforcing the privacy of secret data entrusted to the cloud provider by the cloud user" [Roc10].

Increased vulnerability

Since cloud-based solutions are exposed on Internet, they are more vulnerable to attacks from malicious users and hackers.

3.2.6 Conclusions

Mobile applications testing on cloud is a recent paradigm with lot of potentialities. It enables companies to test their applications on a cloud of real devices. Cloud computing is increasingly becoming the means through which online services are made available but it still has many disadvantages. The major concerns are related to security and availability issues, immediately followed by availability of these services which regards to outages

[24] http://www.javacodegeeks.com/2013/04/advantages-and-disadvantages-of-cloud-computing-cloud-computing-pros-and-cons.html

and downtimes. Vendor lock-in is another big concern that companies have that is preventing them from adopting these services.

3.3 Verdict

With functional testing in mind where one can test applications against the requirements specifications and hence validate them, mobile test automation frameworks arise as a viable solution to the problem in hands. These tools, if used on real devices and in an environment as close as possible to production, can give an elevated degree of confidence.

Cloud-based solutions that specialize in automated tests are a recent paradigm. Based in cloud computing, these architectures are a big evolution on mobile automation but also pose big disadvantages as previously described.

Security concerns[25] are still the major inhibitor of cloud adoption at many large companies.

Self-hosted solutions are still the option that companies prefer, not only because of the security concerns but because of the more comprehensive control that these solutions provide as well.

For this reason, a self-hosted solution was preferable for the job in the hands. The research conducted in the Section 3.1 led to the conclusion of what was the better tool to be integrated in the deployment pipeline within this dissertation.

Calabash comes as the most feature-rich framework. It offers good support not only for iOS but Android and mobile web as well. Regarding iOS support, this frameworks presents itself as the one with better gestures support. Additionally, it offers support for executing tests on devices, one of the main objectives of the solution in mind.

[25] http://www.infoworld.com/t/cloud-security/9-top-threats-cloud-computing-security-213428

Table 3.3: Comparison of Keynote DeviceAnywhere and Perfecto Mobile

Factor	**Keynote DeviceAnywhere**	**Perfecto Mobile**
Installation method		
Installation method	Downloadable software	Web-based
Files management		
Over-the-air (OTA) install		✓
Files management		✓
Cleanup feature		✓
Integration with CI	✓	✓
Operating systems		
iOS and Android	✓	✓
Automation		
Visual objects (image, OCR)	✓	✓
Native objects: Android	✓	
Native objects: iOS		✓
Hybrid object support	✓	
Web objects	✓	Limited
Script portability		✓
Manual testing		
Helper commands (e.g. launch application, send SMS)	✓	✓
Power cycle	✓	✓
Examine device (e.g. CPU, battery)		✓
Real devices and emulators	✓	✓
Retrieve device logs	✓	✓
Screenshot support	✓	✓
Video recording		✓
Install application (OTA and file system)		✓

Chapter 4

Implementation

This section contains a detailed description of the developed solution divided in four section where the first is a simple introduction to better contextualize the evolution of the solution. The other three sections report thoroughly the implementation.

Section 4.1 describes the starting point from where this dissertation started by giving an overview of the early continuous integration solution that was being used by the iOS team at Blip. Section 4.2 exposes the problems of the early pipeline and the solutions that were applied to improve the deployment pipeline. Section 4.3 focuses on the automated acceptance tests and the steps taken to add such testing process to the deployment pipeline. Finally, in the Section 4.4, all the means to provide feedback to the team are reviewed.

4.1 Starting point

At the beginning of this dissertation, the iOS team of Blip had already a mobile development environment as shown in Figure 4.1 which shows a typical CI solution that mainly includes building and unit testing.

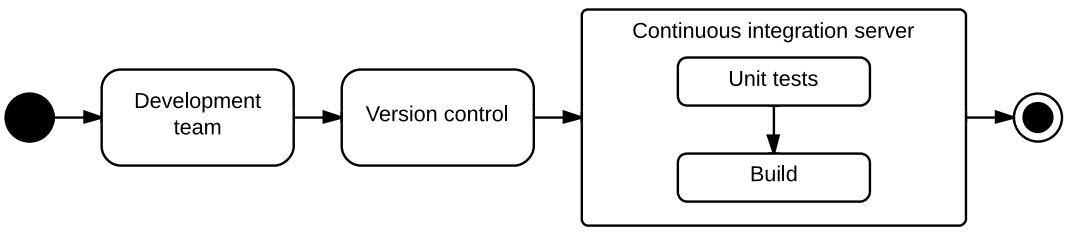


Figure 4.1: Initial solution of continuous integration

For the sake of simplicity and clarity, this solution will be henceforth referred as *deployment pipeline* or simply *pipeline*, even though at this point it still does not meet all the requirements as shown in the Chapter 2.

The pipeline starts with the development team itself. That is to say, when someone commits a change into the code repository, the continuous integration server is triggered.

Version control, or revision control, is the management of changes to collections of information. Many organizations use revision-control software to track and manage the complexity of a project as it evolves. A version control system (VCS) manages *repositories*. "A repository can consist of anything from single file to a complete source tree. In addition to this current representation of its state, a repository contains history. Each repository's internal state is the result of several patches. A patch is a logically connected collection of operations that changes the state of a repository." [LSL07]

The code programmers write changes often. Bugs need to be fixed, features need to be added, and content needs to be changed. Most code is stored as plain old text files, and the code is changed by editing these files. Every time a change is saved, the old version of the file is overwritten with a new one. Unfortunately, no programmer is perfect, and sometimes, mistakes are made. If one makes a change to a file, saves it, compiles it, and finds out that something went wrong, it is often helpful to be able to go back to the old version or to get a report of what was actually changed, in order to focus on what may have gone wrong.[1] Managing versions manually by making copies of files quickly becomes a nightmare. Version control systems are programs designed to handle this.

Still on the topic about version control systems, they are divided into two groups: *centralized* and *distributed*.[2] Centralized version control systems are based on the idea that there is a single *central* copy of the project on a server and programmers will *commit* their changes to this central copy. On the other hand, distributed systems do not necessarily rely on a central server to store all the versions of a project's files. Instead, every developer *clones* a copy of the repository and has the full history of the project on their own hard drive. This copy, or *clone*, has all of the metadata of the original. Distributed VCSs offer several advantages over the centralized systems: most actions are faster because do not require access to the remote server, which is great when working without an Internet connection, and "since each programmer has a full copy of the project repository, they one can share changes with one or two other people at a time to get some feedback before showing the changes to everyone."

The VCS used by the iOS team is Git[3], a distributed VCS, and *gitflow*[4] is the workflow being used in the project that defines a strict branching model designed around the project release which is best-suited for managing larger projects. This workflow assigns very specific roles to different branches and defines how and when they should interact.

[1] http://blogs.atlassian.com/2012/02/version-control-diffs-patches
[2] http://blogs.atlassian.com/2012/02/version-control-centralized-dvcs
[3] http://git-scm.com/
[4] http://nvie.com/posts/a-successful-git-branching-model/

The core idea behind this workflow is that all feature development should take place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the master branch will never contain broken code, which is a huge advantage for continuous integration environments. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Figure 4.2 illustrates the usage of such branches.

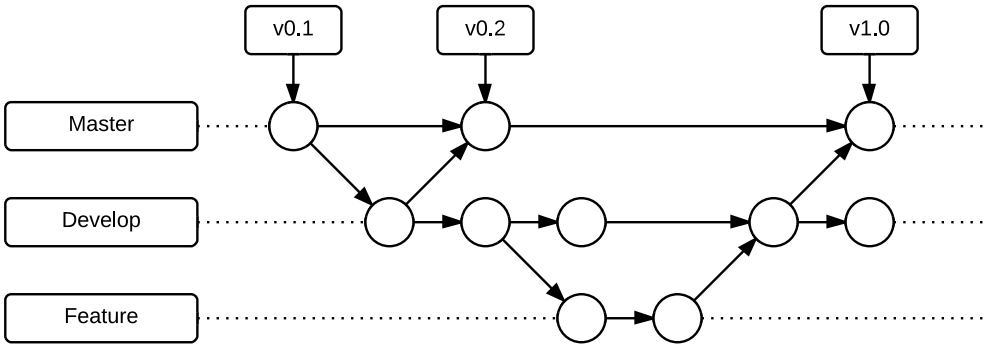


Figure 4.2: Branches usage with the Gitflow workflow

Gitflow uses central repository as the communication hub for all developers. Developers work locally and push branches to the central repository. Instead of a single `master` branch, this workflow also uses another branch to record the history of the project. The `master` branch stores the official release history and the `develop` branch serves as an integration branch for features. All feature development should take place in a dedicated branch which makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. Feature branches use `develop` as their parent branch. When a feature is complete, it gets merged back into `develop`. Features should never interact directly with `master`.

The continuous integration server operates both in the developer and the feature branches. Whenever one of those branches gets updated, GitLab[5]—the source code repository where all the code is kept—triggers the CI server which in turn runs a series of deployment scripts to validate the code. Every time the server is triggered, it runs a sequence of steps as illustrated in Figure 4.1. Jenkins[6] is the continuous integration tool running on the server and it is the core of all the deployment pipeline.

Jenkins is a server-based system that provides continuous integration services for software development. It supports various Source Control Management (SCM) tools, including Git, and can execute Apache Ant[7] and Maven[8] projects as well shell scripts. Jenkins

[5] https://www.gitlab.com/
[6] http://jenkins-ci.org/
[7] http://ant.apache.org/
[8] http://maven.apache.org/

also comes with different mechanisms to trigger builds—they can be triggered by commit in a version control system, when other builds have completed, by requesting a specific build URL or through scheduling via a cron-like mechanism.

On Jenkins, each independent step is called a *job*. For instance, the two steps shown illustrated in Figure 4.1—`Build` and `Unit tests`—could be configured as two separated jobs. Another important concept to retain is that a *build*, in a continuous integration context, is the entire set of all jobs (or steps) needed to obtain the software product (and not just compiling). Jenkins can run multiple jobs at the same time corresponding to the number of *executors* configured and available. This number represents how many concurrent jobs Jenkins will run on the machine where it is hosted. The maximum number of jobs is dependent upon hardware specifications of the server, such as available memory, disk speed, availability of SSD, and overlap of source code. The Jenkins instance running on the illustrated pipeline has two build executors which means it can only executes two jobs at once.

Back to the pipeline, whenever the CI server is triggered, the following sequence takes place:

1. Jenkins launches the `Unit tests` job which in turn:
   (a) pulls last changes from VCS;
   (b) executes shell script that downloads and installs all the external dependencies of the source code;
   (c) executes a shell script that drives unit tests of the application;
   (d) if the unit tests script terminates successfully, Jenkins adds the `Build` job to the build queue; otherwise, Jenkins sends a email to all the teams notifying about the failure of the build.
2. Jenkins launches the `Build` job:
   (a) pulls last changes from VCS;
   (b) executes shell script that downloads and installs all the external dependencies of the source code;
   (c) executes a shell script that builds the application;
   (d) if the build script terminates successfully, as post-action of the step, the resulting executable generated by the build process is uploaded for internal release to TestFlight[9], a platform used to distribute development builds of applications, and delivering the application to multiple team members, namely the Product

[9] TestFlight (http://testflightapp.com/) offers a convenient way to publish iOS applications without publishing them on the Appstore in a automated fashion that enables users to download the latest version continuously.

Owner (PO), Business Analyst (BA), Delivery Manager (DM) and Quality Assurance (QA); otherwise, Jenkins send a email to all the teams notifying the failure of the build.

With regards to this, one must note that these jobs operate in the `develop` branch but their flow is independent from the branch, so at anytime is possible to change the branch from where Jenkins pulls the code and run the pipeline on that same code. In fact, Jenkins had five more jobs with exactly the same steps as the `Build` job but each one was pulling the code from a specific feature branch. The reason to be five jobs was to have each team of the project with a dedicated job to validate their code. All the jobs were triggered on a commit basis. Such practice is imbued with the vision that every commit should build on the integration machine.

Figure 4.3 shows two versions of the same Jenkins view of the pipeline: on the left (4.3a), both jobs have terminated successfully; on the right (4.3b), `Build` job has failed. The order by which the jobs appear is purely alphabetical; it does not represent the order by which they are executed.

A more profound examination of the steps is required to fully comprehend the pipeline implementation at such initial phase and its flaws.

In the first place, looking at the previously listed steps of each job, one can observe some steps in common, namely (a) and (b), which translates to unnecessary repeated behavior on different jobs for supposedly the same codebase. The same code was being pulled twice. Instead of sharing the same workspace, the jobs were using two separated workspaces. This behavior was increasing the total time to validate the build. The CI server should check out changes from the repository only once and, thereafter, build the system and run unit and integration tests on the same source code. Apart from the extra time it was taking, a more severe problem could arise if, between the termination of the `Unit tests` job and the beginning of the `Build` job, one commit or more were made. Such behavior would result in the pipeline not running entirely on the same code and not correctly validating it.

In the second place, the `Unit tests` job was merely running the unit tests without extracting valuable information such as code coverage for example to measure the code quality and raise the confidence level.

(a) Pipeline terminated successfully (b) Build job failed

Figure 4.3: Jenkins view with deployment jobs

In the third place, the `Build` job was not only encompassed with building the project, but uploading the resulting software product to an online platform as well. For this reason, there were times when the job would fail, not because the code was not compilable, but because the upload had failed. For instance, Figure 4.3b only allows to know that the job failed but it is not clear what actually failed: if the build script or the upload; rather, one has to look further into the console log of the job to understand what broke the build.

At last, it is clear the pipeline misses a step with automated acceptance tests, an important confidence indicator, indispensable for a more complete deployment pipeline.

Overall, the early pipeline where this dissertation started from was only on its first baby steps, lacking a well-defined automated solution in order to achieve a continuous delivery solution. It had repeated logic making the pipeline harder to maintain and optimize. The average duration of the pipeline—unit testing, building and deployment—was 20 minutes.

4.2 Deployment pipeline

This section describes the deployment pipeline implemented to reach a solution of continuous delivery. The previous section mentioned some aspects of the early pipeline that needed to be improved. The solution here presented solves all those points.

PROBLEM 1:

Deployment scripts hard to maintain with repeated logic and possibility of inconsistency in the code due to be being pulled twice from the repository.

SOLUTION:

The two jobs were broken into multiple jobs, not only for refactoring and sanitizing the code, but for modularizing the pipeline as well. A pipeline composed by several independent jobs, easy to configure and reprogram, result in a set of actions that can be reorganized at will to fit upcoming needs at anytime. Because each job works as a module, one can execute the pipeline with just a few modules activated. For instance, one may want to only compile and run the unit tests of a build, without triggering neither the automated acceptance tests nor the upload of the executable for internal release.

The modularization achieved through this solution is illustrated in Figure 4.4. The notes in the diagram show what actions are executed in each job. The upper part of the figure shows the initial version of the pipeline while the lower part shows the pipeline after all the optimizations and refactoring done.

The pipeline was broken into four jobs. The process of checkout of the code from the version control system was moved to a job entitled `Pipeline` and the verification and installation of the required dependencies of the project was moved to the `Setup` job. The `Unit tests` and `Build` jobs have lost all the unnecessary and unrelated logic, which was stripped and allocated in the new jobs.

The `Pipeline` job is the entry point of the deployment pipeline and it is the job that gets triggered by the Jenkins whenever the repository receives a new commit. This job differs a little from the rest to the extent that it do not contains Bash shell script (more information on this ahead); rather, it contains a Groovy script to orchestrate the entire flow of the pipeline. It also has a preliminary step that pulls the changes from the VCS.

Having a *master job* responsible for the whole orchestration of the pipeline makes it configurable and more susceptible to changes and improvements. Listing 4.1 shows the flow of the implemented pipeline at Blip[10].

```
1  workspace = build.environment["WORKSPACE"].toString()
2  // Setup
3  job_setup = build("Setup", PROJECT_DIR: workspace)
4  // Unit Tests
5  job_unit_tests = build("UnitTests", PROJECT_DIR: workspace)
6  // Build
7  job_build = build("Build", PROJECT_DIR: workspace)
8  build_folder = job_build.build.environment["WORKSPACE"].toString()
9  ipa = build_folder + "/ipa/application" + job_build.build.number + ".ipa"
10 // Deploy
11 job_deploy = build("Deploy", PROJECT_DIR: workspace, IPA: ipa)
```

Listing 4.1: Pipeline flow in Groovy script

The traditional way of setting up Jenkins to execute jobs in a certain order is over-complicated. For instance, to have the sequence `A`→`B`→`C`, job `A` must be configured to have a post-build action to trigger `B` and `B` must be configured to trigger `C`. By just simply looking to the jobs, it is not clear what triggers what and in which circumstances. Using the Build Flow Plugin[11], it is possible to manage the Jenkins jobs orchestration using a dedicated domain-specific language (DSL), extracting the flow logic from jobs. With such approach, complex build workflows (such build pipelines) are handled as a dedicated entity in Jenkins. Without such plugin, to manage job orchestration one has to combine a lot of plugins, polluting the job configuration and, subsequently, making the build process scattered in all the pipeline jobs and very complex to maintain it.

The flow DSL is built over Groovy, an object-oriented programming language for the Java platform. The Groovy Programming Language runs on top of the Java Runtime Environment. Because Jenkins runs on top of Java and has built-in support for this scripting language, with Groovy scripts it is possible to access all the internals of Jenkins and write more complex scripts.

[10] Some unnecessary code for the example was omitted.
[11] https://wiki.jenkins-ci.org/display/JENKINS/Build+Flow+Plugin

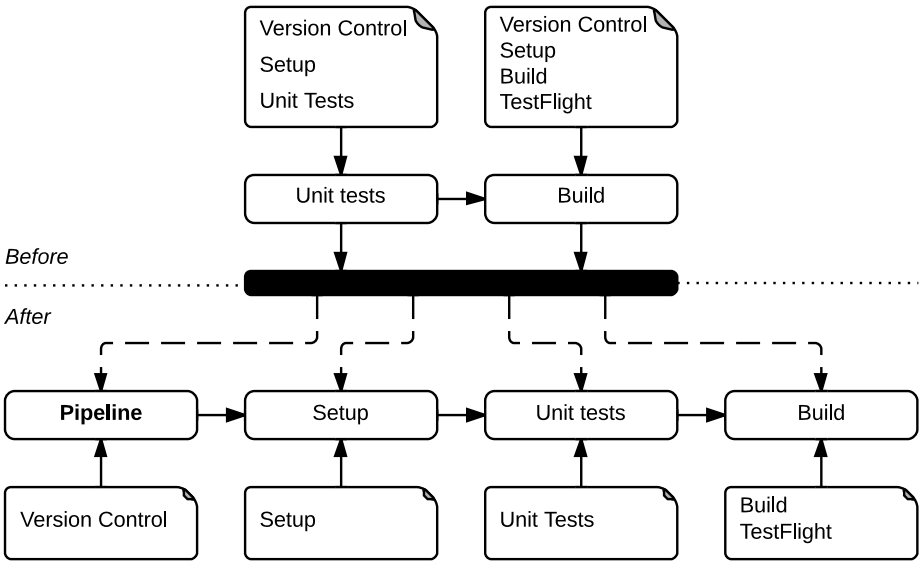


Figure 4.4: Pipeline modularization into independent units

PROBLEM 2:

`Unit tests` job limited to only running the unit tests and not producing valuable metrics to understand the status and evolution of the unit tests.

SOLUTION:

The job at issue was upgraded in order to generate human-readable summary reports. The script to execute the units tests was improved to output the the results in a format understood by `gcovr`[12], a utility that analyses and summarizes the lines of code that are executed—or *covered*—while running an executable. The reports generated by this utility are produced in machine readable XML, in Cobertura format, and with the assistance of a Jenkins plugin, it is then converted to a human-readable HTML report.

PROBLEM 3:

`Build` job not only responsible for compiling the project but uploading it to a platform on the web as well which, in situations of the job failing, it was not clear the reason.

SOLUTION:

The job was split into two: one for compiling the project, the other for uploading to TestFlight for internal release. This solution follows the philosophy of the first problem which intent is to keep the pipeline modular as much as possible. Figure 4.5 evidences the division of the `Build` job into `Build` and `Deploy`. With such separation, the result of the execution of the former will always reflect the compilation result, when previously, the cause could also be a problem with the upload to TestFlight, leading to unnecessary

[12] http://gcovr.com

doubts without further investigation. The latter uploads the resulting software product of the `Build` to TestFlight.

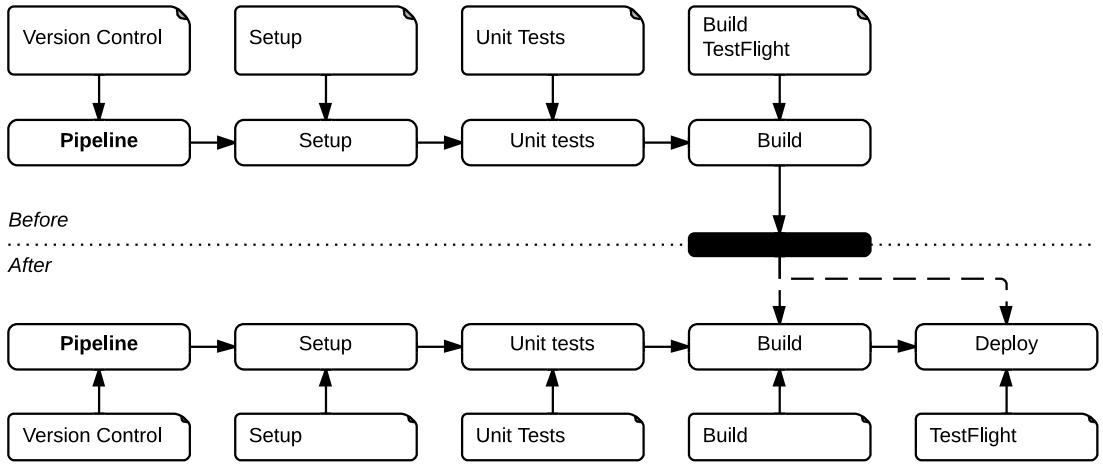


Figure 4.5: Second iteration of the pipeline modularization

PROBLEM 4:

Pipeline does not count with a step of automated acceptance tests which is mandatory in order to obtain a solution of continuous delivery.

SOLUTION:

Ideally, the automated tests should be part of the deployment pipeline. The intended was to have this step between the build and deployment phases, that is to say, deploy the software product only if the automated acceptance tests passed.

A new job for this purpose was created and added to the pipeline but soon it proved to be inefficient. With a few tests written, the job was already taking more than 20 minutes to run all the tests in the iOS simulator, which is the same amount of time that the rest of the entire pipeline was previously taking. Such approach turned out to be inviable, mainly because of the pipeline being triggered on a commit basis. Chapter 5 describes more in detail the technical difficulties found regarding this problem.

As a result, automated acceptance tests were not included directly in the pipeline. They are executed in parallel, not on a commit basis, but in determined periods of time. Moreover, the flow implemented is not exactly as the shown in Figure 1.1 in the Problem Statement. The initial idea was to run the tests on the iOS simulator and, only if they passed, then the tests would be executed on physical devices. Again, such approach proved to be impractical because the tests on physical devices were much slower than on the simulator, taking almost twice the time to finish.

Two jobs were created to focus entirely on automated acceptance tests. Figure 4.6 shows these two new jobs next to the pipeline but not as part of it.

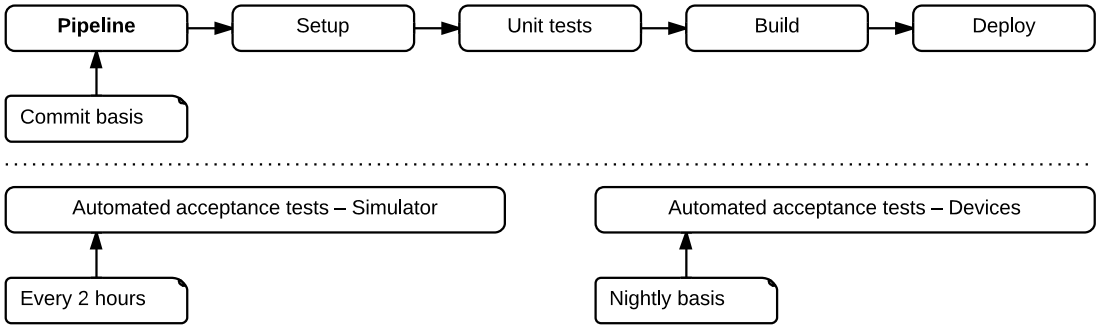


Figure 4.6: Improved pipeline with automated acceptance tests

Jenkins supports the *master/slave* mode where the workload of building projects can be delegated to *slave* nodes, allowing a single Jenkins installation to provide different environments needed for builds and tests[13]. A *master* operating by itself is the basic installation of Jenkins and in this configuration the master handles all task for the build system. Installing a slave allows to free up master resources. Using Jenkins with just a master consumes a lot of resources (e.g., memory and CPU) and that is where setting up a slave to pick up the load is a good option. A slave is a computer that is set up to offload build projects from the master and once setup this distribution of tasks is fairly automatic. The exact delegation behavior depends on the configuration of each project: some projects may choose to "stick" to a particular machine for a build, while others may choose to roam freely between slaves. Using a well established virtualization infrastructure, it is also possible to run multiple slave instances on a single physical node.

The jobs created for automated acceptance tests were configure to only run on a slave (another computer). Thus, the build queue of the master is never occupied with these two jobs that take a long time to complete. Furthermore, such tests have a high resource footprint and running these jobs on the master was way too error prone due to performance issues.

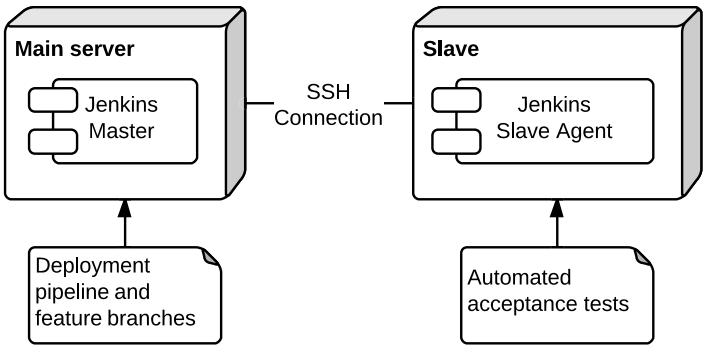


Figure 4.7: Jenkins nodes and respective functions

[13] https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds

Figure 4.7 illustrates the architecture of the solution implemented using the master/slave mode of Jenkins. All work related with automated acceptance tests is done in the slave to spare resources in the master node. A more detailed vision about the acceptance tests follows in the next section.

4.3 Automated acceptance tests

The presented solution in this section is a group of proceedings allied with some tools that result in a system that automates all the mobile testing in a continuous deployment manner running functional tests, with special emphasis in a range of physical devices. This set of steps is part of the deployment pipeline as previously demonstrated in Figure 4.6.

Preferably, the execution of the automated acceptance tests would be a job part of the main pipeline, described in the previous section. This way, the tests would run in the same codebase as the rest of the pipeline. Due to time constraints because of the nature of the pipeline of being executed on a commit basis, such approach proved to be inviable.

As noted several times along Chapter 1, testing in real devices was a big goal and almost mandatory. Having a great device coverage increases a lot the confidence level on the system under test. It lets one believe that the application will work as expected in the production environment without requiring much manual testing which is slow and expensive.

This solution tests the applications in two different environments: on the iOS simulator and on physical devices. The current implementation is only testing the application in the iPad simulator and in a iPad respectively.

Calabash proved to be the tool of choice to conduct the acceptance tests after the literature review described in the Chapter 3. Even if Blip currently only aims to use this framework on iOS devices, Calabash can run the same tests with few or no changes on Android devices which makes it a good bet in the long term.

Both jobs have similar flows and they are kind of a merge of the `Pipeline`, `Setup` and `Build` jobs. In their core, both jobs of automated acceptance tests pull the last changes from the VCS and verify the dependencies, installing or updating the necessary ones. The build step differs a little from the `Build` job in the way that it compiles the application under test using a special scheme[14] so that the Calabash framework is linked together with the generated executable. The remaining flow differs from one job to another. A description for each follows.

Tests on the simulator

After building the application, the script ensures that the simulator is ready to receive the application and drive the tests on it. It enables the accessibility of the simulator, a

[14] An Xcode scheme defines a collection of targets to build, a configuration to use when building, and a collection of tests to execute.

prerequisite of Calabash so it can have full control of the application.

Calabash is then invoked to launch the application on the simulator, and as soon the application is launched, it starts executing the Cucumber features.

Tests on the devices

After building the application, to distribute it to the devices, it is necessary to undergo a process of archiving and validating the application. Xcode archives allows to store the built software product along with critical debugging information, in a bundle. Archiving the debugging information makes it easier to interpret crash reports.

Before resuming the description of the script, some concepts about the distribution of an iOS application must be introduced. In order to distribute an application, one should:

1. Register all test devices
2. Create a distribution certificate
3. Create an ad hoc provisioning profile
4. Archive and validate the application
5. Create an iOS App Store Package

An *ad hoc provision profile* is a distribution provisioning profile for iOS applications that allows an application to be installed on designated devices and use key technologies and services without the assistance of Xcode. It is one of the two types of distribution provisioning profiles one can create for iOS applications. The other type is used to submit the application to the store. Each iOS device in an ad hoc provisioning profile is identified by its Unique Device Identifier[15] (UDID). Figure 4.8 shows the constitution of an ad hoc provision profile.

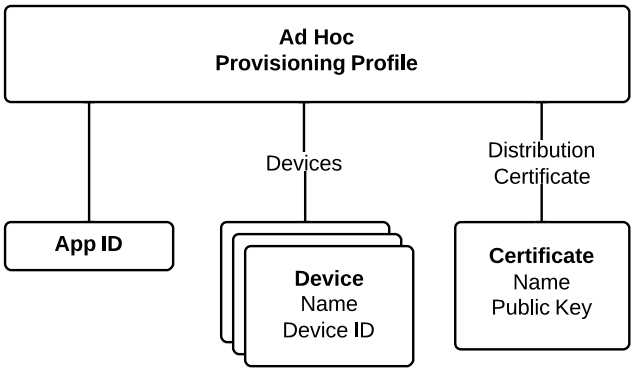


Figure 4.8: Constitution of an ad hoc provisioning profile

[15] A 40-character long hexadecimal value that uniquely identifies a device.

Getting back to the point, to install the application on devices, an *iOS App Store Package* must be created. The script also takes this into account and generates an iOS App Store Package (a file with a `.ipa` filename extension) from the archive previously mentioned. In this step, the `.ipa` gets signed with the distribution certificate specified in the ad hoc provisioning profile.

Afterwards, the script finds which devices are connected to the computer—the slave agent, in this case—in the form of a list of pairs. Each line of that list defines a device which is represented by its type—`iPhone` or `iPad`—and its UDID. Listing 4.2 shows a possible output of the script running in a computer with two connected devices. If no devices are connected to the computer, the Jenkins job aborts its execution.

```
1 iPad 9c1048b25e6a49d3ae20a041c9d5d98e8224078a
2 iPhone 01ead2bb47ca4c71a682417156b1f47dd3c65bc8
```

Listing 4.2: Two connected devices

The remaining steps live in a loop that iterates over each connected device. The following logic is applied to each device.

To start with, knowing the Wi-Fi address of the device is mandatory. For that effect, the script searches the address of the connected device in an associative array (manually mapped) that associates the UDID of multiple known devices to their respective Wi-Fi addresses. Knowing this address is mandatory because it is used by Calabash in order to exchange commands with the device.

Then, a script is used to deploy and install the application on the device, Next, Calabash is invoked just like in the job for the simulator, though there are now two more arguments necessary: the device ID and its Wi-Fi address. These arguments tell Calabash that it must execute the tests on a device. In spite of the necessity of specifying the Wi-Fi address, the device must be connected via USB.

Algorithm 4.1 shows the algorithm applied to the Jenkins job that runs the automated acceptance tests on devices. Since the written tests in Cucumber are only really fully working for iPad at the moment of writing and only one physical device is currently being used (one iPad), the current implementation, although prepared for multiple devices, is running the tests in series. Chapter 5 contains a more profound explanation on this subject, including a solution that drives tests on devices in a parallel fashion.

Algorithm 4.1: Automated acceptance tests on devices

Require: *sourceCode* ▷ Most updated source code

```
1:  build ← Build(sourceCode)
2:  ipa ← PackageApplication(build)
3:  devices ← GetDevices()
4:  addresses ← LoadAddresses()
5:  if Empty(devices) then Exit(1)                    ▷ Abort execution
6:  end if
7:  for i ← 0, Count(devices) do
8:      deviceType ← Type(devices[i])
9:      deviceId ← Id(devices[i])
10:     deviceAddress ← addresses[deviceId]
11:     Deploy(deviceId, ipa)                         ▷ Install on device
12:     Calabash(deviceType, deviceId, deviceAddress)
13: end for
```

4.4 Feedback

Feedback plays a big role in the world of continuous delivery. Improving it allows one to identify problems, and so resolve them, as early in the process as possible [HF10]. CD aims to make frequent, automated releases of software and so feedback is essential to such environment. According to [HF10], there are three criteria for feedback to be useful:

- Any change, of whatever kind, needs to trigger the feedback process.
- The feedback must be delivered as soon as possible.
- The delivery team must receive feedback and then act on it.

Everybody involved in the process of delivering software should be involved in the feedback process. That includes developers, testers, operations staff, infrastructure specialists and managers. Being able to react to feedback also means broadcasting information through the use of visible dashboards and other notification mechanisms so "that feedback is, indeed, fed-back".

That said, the solution of continuous delivery implemented within this dissertation did not neglect such aspect. Two mechanisms are used to give feedback to the teams: Jenkins itself with a view specially designed for the deployment pipeline and a notification system via email.

The default view of Jenkins shows a simple list of jobs with some pertinent information but not enough. The most important visual indicator is the status ball that represents the status of the last build—*green* in case of success; *red*, otherwise. It lets one to have a quick perception of a job status.

Since the starting point of the deployment pipeline, its view on Jenkins has suffered many improvements. Figure 4.9 shows this view back then and the current one as well. The initial deployment pipeline at Blip had a basic view as shown in the Figure 4.9a. The main jobs of the pipeline (`Unit tests` and `Build` which operate over `develop`) were mixed with the jobs operating over the feature branches. Their order in the view was purely alphabetical.

Figure 4.9b represents the current state of the deployment pipeline view. The first big difference is the division between the main pipeline (*Pipeline* section) and the team jobs (*Teams* section). There is also two new sections corresponding to the new jobs of automated acceptance tests (*Automation - Simulator* and *Automation - Devices* sections).

Pipeline

This section contains the improved deployment pipeline. Jobs are sorted, not by their name, but by the order in which they are executed in the pipeline. Such approach improves the readability and accelerates the feedback process.

Teams

This section simply contains the jobs that operate in the feature branches of the teams. The only difference in these jobs is the branch they pull from. Beyond that, they simply build the application.

Automation - Simulator

Section for automated acceptance tests on simulator. Jobs on this section are executed on a slave node.

Automation - Devices

Section for automated acceptance tests on devices. Like the previous section, jobs on this section are also meant to be executed on a slave node.

This new view is always being displayed on multiple big screens for the purpose of giving continuous feedback to the teams. It is also accessible by anyone who wish to open the Jenkins view on the browser.

The deployment pipeline also counts with a new view, shown in Figure 4.10. This views allows one to have a clear vision of the pipeline flow and detect if something is breaking the process.

The other mechanism of feedback being used on the solution is the notifications via email in case of failure of any job. Every single job is configured to broadcast an email if something fails. As a result, all the teams receive instant feedback about malfunctions in the pipeline.

S	W	Name ↓	Last Success	Last Failure	Last Duration	
		Team A	4 days 11 hr - #268	N/A	7 min 11 sec	
		Team B	11 days - #687	N/A	15 min	
		Team C	4 days 11 hr - #170	N/A	17 min	
		Build	19 days - #1346	1 mo 2 days - #1337	11 min	
		UnitTests	19 days - #589	N/A	18 min	
		Team D	18 days - #485	1 mo 6 days - #481	8 min 42 sec	
		Team E	28 days - #195	N/A	15 min	

(a) View at starting point

InHouse - Pipeline

S	Name	Last Success	Last Failure	Last Duration
	Pipeline	7 days 15 hr - #288	18 days - #271	14 min
	Setup	7 days 15 hr - #191	N/A	13 sec
	UnitTests	7 days 15 hr - #379	18 days - #361	3 min 27 sec
	Build	7 days 15 hr - #1439	N/A	3 min 50 sec
	Deploy	7 days 15 hr - #105	27 days - #45	5 min 19 sec

Teams

S	Name	Last Success	Last Failure	Last Duration	
	Team A	4 days 11 hr - #268	N/A	7 min 11 sec	
	Team B	11 days - #687	N/A	15 min	
	Team C	4 days 12 hr - #170	N/A	17 min	
	Team D	18 days - #485	1 mo 6 days - #481	8 min 42 sec	
	Team E	28 days - #195	N/A	15 min	

InHouse Automation - Simulator

S	Name	Last Success	Last Failure	Last Duration	
	Automation - Simulator	2 min - #455	2 hr 2 min - #454	47 min	

InHouse Automation - Devices

S	Name	Last Success	Last Failure	Last Duration	
	Automation - Device	1 day - #183	4 hr 19 min - #184	10 sec	

(b) Current view

Figure 4.9: Jenkins view of the deployment pipeline: back then and now

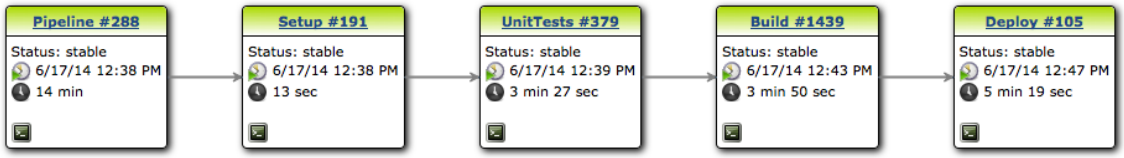


Figure 4.10: Build graph of the deployment pipeline

The mechanisms presented so far give an interesting overview of the overall pipeline. The view of Figure 4.9b exposed on big visible displays which aggregate information from the build system provide high-quality feedback. At any time, just by quickly looking to a display, one can known if the build is green or not. Conversely, each job has detailed and customized views with more information relatively to its intrinsics.

To start with, a common view of all the jobs is the *console output*. The console output allows to check all output generated the by the job, whether by the Bash scripts or pre-actions and post-actions. This view gives vital pieces of information, whether one wishes to find the root of the problem of a job failing or simply to understand what is happening behind the scenes.

Then, some jobs have also special views. `Unit tests` job has two extra views, one that consists of a graph showing the evolution of the code coverage and a second one consisting in a detail report. Figure 4.11 shows the code coverage graph that allows to have a quick perception on the code coverage evolution over time.

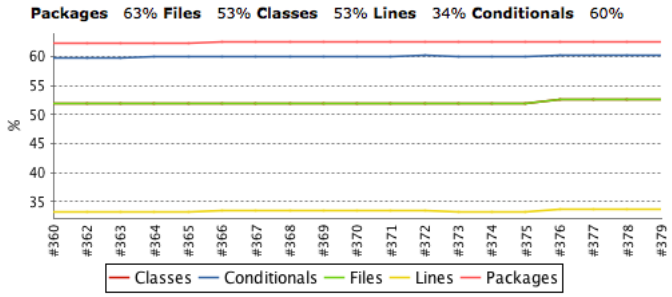


Figure 4.11: Code coverage graph

A code coverage report is also generated which shows the code coverage for each package in the application. It also allows to go deeper and see the code coverage (or lack thereof) for an individual class. In this level, Jenkins displays both the overall coverage statistics for the class and also highlights the lines that were executed in green and those that were not in red. Figure 4.12 shows a example of code coverage report of a class.

The jobs of automated acceptance tests also have a special view with a detailed report in order to show what tests have failed.

When Cucumber is executed, it generates a report that verifies whether or not the software behaves the way the Gherkin document says.

File Coverage summary

Name	Classes	Lines	Conditionals
BFEEventPageModulesTransformer.m	100% 1/1	87% 33/38	66% 21/32

Coverage Breakdown by Class

Name	Lines	Conditionals
BFEEventPageModulesTransformer_m	87% 33/38	66% 21/32

Source

BetfairExperience/BetfairExperience/Classes/Controllers/Features/EventPage/Configuration/BFEEventPageModulesTransformer.m

```
 1     //
 2     //  BFEEventPageModulesTransformer.m
 3     //  BetfairExperience
 4     //
 5     //  Created by Rui Teixeira on 3/4/14.
 6     //  Copyright (c) 2014 Blip. All rights reserved.
 7     //
 8
 9     #import "BFEEventPageModulesTransformer.h"
10     #import "BFTypes.h"
11
12     #define NumFromInt(i) [NSNumber numberWithInt:i]
13     #define NumFromString(s) [NSNumber numberWithInt:[s integerValue]]
14
15     @implementation BFEEventPageModulesTransformer
16
17  4  + (NSDictionary*)eventPageModuleFromDictionary:(NSDictionary*)dict {
18
19  4      NSArray *moduleNames;
20  4      NSMutableDictionary *modulesByEventTypes = [[NSMutableDictionary alloc] init];
21
22 30      for (NSString* eventTypeKey in dict) {
23
24 11          moduleNames = dict[eventTypeKey];
25
26 11          [modulesByEventTypes setObject:
    [BFEEventPageModulesTransformer modulesFromModuleNames:moduleNames] forKey:NumFromString(eventTypeKey)];
27 30      }
28
```

Figure 4.12: Code coverage report

Gherkin is the language that Cucumber understands. It is a Business Readable, Domain Specific Language[16] that lets one describe software's behavior without detailing how that behavior is implemented. Gherkin serves two purposes: documentation and automated tests.

Gherkin is a line-oriented language that uses indentation to define structure. Line endings terminate statements. The parser divides the input into features, scenarios and steps. A Gherkin source file usually looks like the example in the Listing 4.3.

```
1 Feature: Descriptive text of what is desired
2   Textual description of the business value of this feature
3
4   Scenario: Some determinable business situation
5     Given some precondition
6       And some other precondition
7      When some action by the actor
8       And some other action
9      Then some testable outcome is achieved
```

[16] http://martinfowler.com/bliki/BusinessReadableDSL.html

```
10
11   Scenario: A different situation
12       ...
```

Listing 4.3: Example of a Gherkin source file

Cucumber can report results in several different formats, using *formatters*[17]. Some formatters generate files while others print to `STDOUT`. Cucumber allows the use of multiple formatters. The main formatters are:

- **Pretty:** Prints the gherkin source with additional colors and stack traces for errors.
- **HTML:** Generates a HTML report handy for publishing.
- **JSON:** This report contains all the information from the gherkin source, with additional results for each step, including embedded screenshots.
- **JUnit:** This report generates XML files as defined by Apache Ant's `junitreport`[18] task. This XML format is understood by most Continuous Integration servers, who will use it to generate visual reports, and Jenkins is not exception.
- **Rerun:** This report generates a file that lists the location of failed Scenarios. This can be picked up by subsequent Cucumber runs, allowing only previously failed Scenarios to be rerun.

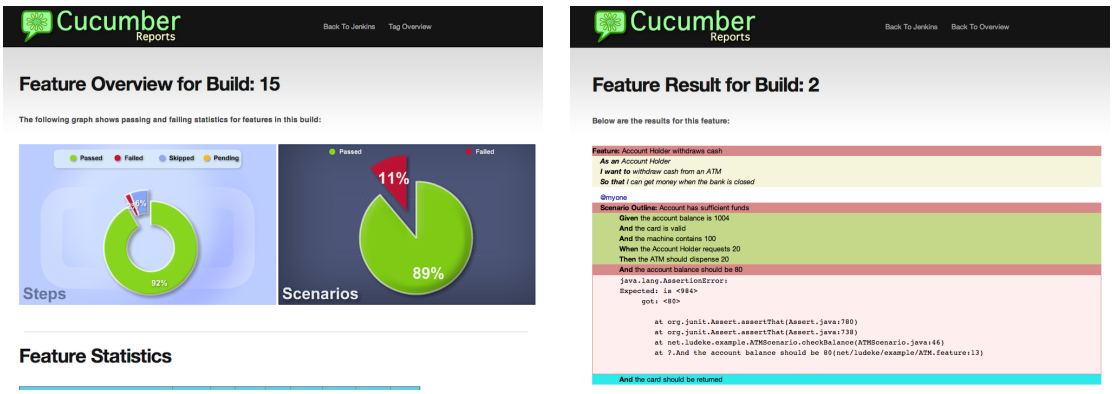
A more profound explanation about the used formatters is done in Chapter 5.

The jobs in the Jenkins are using three formatters. Pretty for printing the results to the console output of Jenkins, JSON to be post-processed into another visual format by a Jenkins plugin and, lastly, Rerun for repeating only the failing scenarios because they are very time-consuming.

The JSON output is used with Cucumber Reports[19] plugin which in turn publishes the results as human-readable HTML reports.

Figure 4.13 shows the two main views of the plugin. On the left, one can view a feature overview page that shows passing and falling statistics for features in the build (Figure 4.13a). On the right, one case see a more detailed view of a specific feature (Figure 4.13b). The plugin highlights the scenario steps that were successfully executed in green and those that failed in red.

[17] http://cukes.info/reports.html
[18] https://ant.apache.org/manual/Tasks/junitreport.html
[19] https://github.com/masterthought/cucumber-reporting



(a) Feature overview (b) Feature detailed view

Figure 4.13: Main views of Cucumber Reports

Chapter 5

Experiments

The implementation described in the Chapter 4 was the result of a long period of experiments. This chapter describes the road traveled to reach the final solution.

Section 5.1 describes the first steps taken to develop a solution according to the dissertation objectives, a process whose progress took place in a local controlled environment. Then, Section 5.2 explains the migration of the implementation from the local environment to the CI server. Lastly, Section 5.3 reports the optimizations done on the deployment pipeline of the CI solution.

5.1 Local controlled environment

There was a initial phase of experimentation before making changes directly on the CI server to avoid breaking the normal behavior. For this reason, such experiments were done in a local controlled environment, that is to say, another computer. Since the CI server is used to build and test iOS applications, the computer where Jenkins is installed runs on top of OS X Mavericks, a Unix-based OS developed by Apple Inc. This local controlled environment had installed the same operating system (OS).

Initial setup

First step was to install a clean version of Jenkins. The installer creates a system *launch daemon* that launches Jenkins when the machine boots. A daemon is a program that runs in the background as part of the overall system (that is, it is not tied to a particular user) handling requests without direct control of an interactive user. A daemon cannot display any graphical user interface (GUI); more specifically, it is not allowed to connect to the window server. In contrast, an agent is a process that also runs in the background but on behalf of a particular user. Agents are useful because they can do things that daemons can not, like reliably access the user's home directory or connect to the window server. The

difference between an agent and a daemon is that an agent can display GUI if it wants to, while a daemon can not. The difference between an agent and a regular application is that an agent typically displays no GUI (or a very limited GUI). OS X uses an utility called `launchd` to launch and control *daemons* and *agents*.

Using Jenkins as a launch daemon posed as a problem because some wanted steps in the deployment pipeline had (or would have) graphical interfaces. For instance, to execute iOS unit tests, the Xcode invokes the iOS simulator. At the same time, the CI server was having problems running the unit tests. Some times Jenkins could not create a instance of the iOS simulator. It turned out that due to a misconfiguration on how Jenkins were configured to launch, some times it had not access to the window server. Both in the server and in the computer, Jenkins was configured as a launch agent instead of a launch daemon.

Xcode was also installed since it was the most indispensable tool for iOS development. Aside this, the Xcode Command Line Tools were also installed[1]. This package enables UNIX-style development via Terminal by installing command line developer tools, as well as Mac OS X SDK frameworks and headers. Many useful tools are included, such as the Apple LLVM compiler, linker and Maker. Command line tools are essential for automation. These tools allied with scripting languages can be used to automate almost anything.

With Jenkins properly configured and Xcode duly installed, the initial setup of the computer was completed.

Exploring Calabash

At this moment, the focus of the dissertation was still just upgrading the continuous integration solution with a new step with automated acceptance tests, inasmuch as exploring the Calabash framework was the next logical step. Moreover, at this stage Jenkins was not used at all because the primary objective was to understand the framework and how to integrate it with the iOS project. As progress was being made, all the steps taken were documented in order to reproduce them later.

The installation of Calabash requires to have Ruby[2] installed on the system. For this purpose, Homebrew[3] and RVM[4] were installed before installing Calabash. Homebrew is a package manager for OS X and RVM a command-line tool which allows one to easily install, manage and work with multiple ruby environments. With Homebrew and RVM fully operational, it was time to install the required gems. A gem at its most basic form is a package and contains code (including tests and supporting utilities), documentation and a `gemspec` (a file that contains information about the gem, such as the gem's files,

[1] https://developer.apple.com/downloads/index.action
[2] https://www.ruby-lang.org
[3] http://brew.sh/
[4] https://rvm.io/

test information, platform, version number and author's name and email)[5]. Calabash is a gem—called `calabash-cucumver`— and it was installed after installing the previous programs.

With the OS configured and Calabash properly installed, it was time to study the inner workings of the framework and explore how to integrate it with iOS projects.

Calabash iOS consists of two parts: a client library written in Ruby and framework (`calabash.framework`), a server framework written in Objective-C. The server framework starts an HTTP server inside the application when the latter is launched that listens for requests from the client library. Figure 5.1 illustrates the overall flow of Calabash.

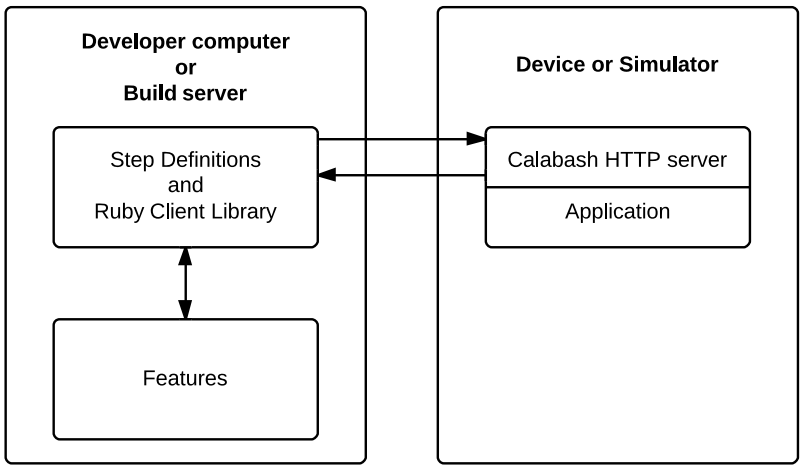


Figure 5.1: Communication between the client library and the server framework

To integrate Calabash with an application, it should be linked with the framework `calabash.framework`. The first attempt of integration of the framework was done with a vanilla project[6]. Three objectives were to be fulfilled in the following order:

Link successfully Calabash with the project

This objective had only one intention: link the framework with the vanilla project in order to understand the process of integration. To understand if the integration was well done and the `calabash.framework` is being loaded, one have simply to verify the log output when executing the application and check if it contains the output of the Listing 5.1.

[5] http://guides.rubygems.org/what-is-a-gem/
[6] https://github.com/calabash/calabash-ios-example

```
... Creating the server: <LPHTTPServer: 0x1462bd00>
... Calabash iOS server version: 0.9.167
... simroot: (null)
... Started LPHTTP server on port 37265
```

Listing 5.1: Log output of an application linked with Calabash

Drive some tests on the simulator

With the framework successfully linked with the application, launching the application through Calabash was the next logical step. For this purpose some very simple tests were written to confirm if Calabash was working correctly. Running the command in Listing 5.2, Calabash setups a features directory for the tests.

```
$ calabash-ios gen

----------Question----------
I'm about to create a subdirectory called features.
features will contain all your calabash tests.
Please hit return to confirm that's what you want.
---------------------------

<RETURN>

----------Info----------
features subdirectory created.
Try executing

cucumber

---------------------------
```

Listing 5.2: Calabash setup of a features directory

Calling `cucumber`, after the previous step, launched the iOS simulator with the application and executed the sample test.

Find the best method of integration

With Calabash linked with the application and the tests being executed, it was time find the best method of integration with any project. The recommend way by the Calabash developers, which was the one used on the first objective, was not the most practical, specially for big projects.

Their method was to create a new target by duplicating the production target in

Xcode. A target specifies a product to build and contains the instructions for building the product from a set of files in a project or workspace[7]. A target defines a single product; it organizes the inputs into the build system—the source files and instructions for processing those source files—required to build that product. Projects can contain one or more targets, each of which produces one product.

The new Calabash target had then to be modified to link with two frameworks: `calabash.framework` and Apple's `CFNetwork.framework`. The latter framework is a dependency of Calabash.

However, this method poses a problem. Whenever a file is added to the main target, it must also be added manually to the Calabash target. This leaves room for mistakes and makes it harder to maintain the project. Through some research and after many tries and tweaks, a better solution was found.

The new solution involved moving the previously linker options from the target scope to a *scheme* scope.

Before proceeding, four Xcode concepts need some clarification:

Workspace
: Contains one or more projects, usually related to one another.

Project
: Contains code and resources.

Scheme
: Defines what happens when one choose an action in Xcode (*Run*, *Test*, *Profile*, *Analyze* or *Archive*). Each target has at least one scheme. In other words, it represents a collection of targets that are used when one chooses on of these actions.

Target
: Defines a list of build settings for a project. It also defines a list of classes, resources and custom scripts to include or use when building the project.

By way of example, a typical and simple approach consists of having one single scheme, which uses the main application target for the *Run*, *Profile*, *Analyze* and *Archive* actions and a unit test target for the *Test* action. Another example is to have two schemes, when building two related applications, that use the same unit tests target but different application targets.

Instead of having a separated target for Calabash, only the main target is used. A new scheme was created but it uses the same target has the production scheme. The difference is that this new scheme overrides the target build settings by changing the linker options to add the required frameworks by Calabash.

[7] https://developer.apple.com/library/ios/featuredarticles/XcodeConcepts/Concept-Targets.html

Testing with a device

Testing with devices was an important objective from the beginning. At this stage, it mattered to understand how to run Calabash on devices. Finding an automated way to install iOS applications on devices was also an objective at this stage.

Deploying an iOS application to a device differs a little from when simply building for the iOS simulator. As described in the Section 4.3, it formerly requires to have the device registered on the Apple's iOS Dev Center[8] and a provision profile associated with the device. To run an application on a device during development, the device must be connected to the Mac.

At this stage, a finding emerged. Wireless deployment was not possible; that is to say, the devices should be connected to the computer in order to install applications on them. Such discovery was disappointing because one of the goals was to entirely develop an wireless solution. The devices for testing purposes ideally would be wirelessly connected to a network eliminating the need to have them physically in the same place but due to technical limitations found throughout the research such scenario was not possible. Further on, Calabash revealed itself unable to work remotely with devices, requiring the devices connected via USB.

Executing tests on a physical device with Calabash iOS was quite simple but during this stage Calabash was suffering some technical changes to work with the recent releases of Xcode 5 and iOS 7 (both products featured new functionalities that either were not supported by Calabash or that broke the expected behavior of the framework). In addition to this, the available documentation did not reflect the last developments of the framework. In order to launch Calabash on a device, two requisites need to be met:

- The application should be installed on the device and be properly linked with Calabash iOS.
- Wi-Fi must be enabled on the device and its IP address should be known prior to the tests. The device ID must also be known.

Then, Cucumber must be called with two extra arguments to specify which device must be used, through its ID, and its Wi-Fi IP address. Listing 5.3 illustrates the command to run Calabash on a physical device.

```
1 $ DEVICE_ENDPOINT=http://<IP>:37265 \
2   DEVICE_TARGET=<UDID> \
3   cucumber
```

Listing 5.3: Invocation of Calabash on a physical device

[8] https://developer.apple.com/devcenter/ios/index.action

Besides these two extra arguments, the experience of testing on devices is identical to the one when testing on the simulator. The results are returned in the same fashion as with the simulator. The only disadvantage found when testing on the device was the extra time it took to Calabash to communicate with the application.

Having Calabash working with a physical device, it was time to find a way to deploy an `.ipa` file to a device without using the Xcode GUI. A command line tool was the perfect candidate for the job. After searching and using some tools, the decision was to use Transporter Chief[9], a Ruby script that allows to deploy an application directly to a USB-connected iOS device in an automated way without using Xcode. To put the Transporter Chief to work, it is necessary to use a library called `fruitstrap`. The downside of Fruitstrap[10] is that it calls one of Apple's private APIs to deploy to the devices which means this API is subject to unannounced changes and may break at some time in the future. Yet, it seemed the best solution available and the most mature.

The process of deploying an application is very straightforward. Listing 5.4 shows the script usage.

```
1 $ ./transporter_chief.rb --device <UDID>
```

Listing 5.4: Transporter Chief usage

At the end of this stage, in theory, all the needed instruments were assembled. It was time to move to Jenkins.

5.2 Calabash integration with Jenkins

After all the experiments on a local environment, it was time to implement the studied workflow on Jenkins.

The deployment script

A new job was created on Jenkins with the intention of being part of the pipeline later. To describe the job workflow, a Bash script was written.

The intended workflow was simple, as previously illustrated in the Algorithm 4.1:

1. build application with Calabash scheme;
2. generate an iOS App Store Package (`.ipa` file);
3. get a list of connected devices;
4. deploy the application on the devices;

[9] http://gamua.com/blog/2012/03/how-to-deploy-ios-apps-to-the-iphone-via-the-command-line/
[10] https://github.com/ghughes/fruitstrap

5. call Calabash on each device.

At this stage, it was decided that tests on simulator were not necessary since it would take unneeded extra time.

For the two first steps of the job, code of existing scripts on the deployment pipeline were reused.

To get a list of connected devices, `idevice_id` was initially used. It is a command line tool, from the libimobiledevice[11] library, that returns the UDID of all connected devices. But, unlike the output illustrated in the Listing 4.2, this utility did not return the device type. Later on, this utility was replaced by a script with some tweaks in order to obtain the device type.

Adding the deployment step was way more uncomplicated since it only involved invoking Transporter Chief with the device ID. Later, there was a time where this step was failing without obvious cause. It required a serious amount of troubleshooting and in the end it turned out the archiving and code signing script was, due to misconfigurations on the project, pulling from the Keychain[12] the wrong provision profile which resulted on errors with the deployment.

Because one of the requisites to run Calabash with devices was to know the Wi-Fi IP address of the device under test, a method to obtain the IP was required. After some research and several experiments, it was concluded that there was no way to obtain dynamically the Wi-Fi address directly from the device because of the security layer on the devices. The solution was to have a hardcoded map[13] with the Wi-Fi address (*value*) associated to the respective device ID (*key*). The deployment script after getting the device ID from the Transporter Chief, looks up the ID on the map. If the map does not have the device ID, the script terminates breaking the job.

A dynamic solution that did not require a hardcoded map was developed but since the set of connected devices to Jenkins was known, it ended up not being used. This solution consisted of two parts: a server in Ruby and a simple iOS application. These two programs use sockets as endpoints for a bidirectional communication and its usages is straightforward: launch the server in the background; deploy application on iOS device and launch it; the application reads the device Wi-FI address, sends that information to the server (through a socket) and terminates the application itself; the server that is listening, in turn, upon receiving the message, outputs it to the console and terminates itself too. With this solution, it is possible to know the Wi-Fi IP address of any iOS device USB-connected.

[11] http://www.libimobiledevice.org/

[12] Apple Inc.'s password management system in OS X. Provisions profiles are also stored in the Keychain.

[13] In computer science, a map or an associative array is an abstract data type composed of a collection of $(key, value)$ pairs.

The final, and the most important, step of the deployment script was executing Calabash on the devices. At this time, with all the previous information gathered, the code resumed to call Calabash as illustrated in the Listing 5.3.

At this time, the essential workflow of the script was ready. Some tweaks and configurations at the OS-level were required when executing the job for the first few times to solve issues like authorization dialogs asking the user credentials to allow access to the Keychain or to use the Xcode Developer Tools (Figure 5.2).

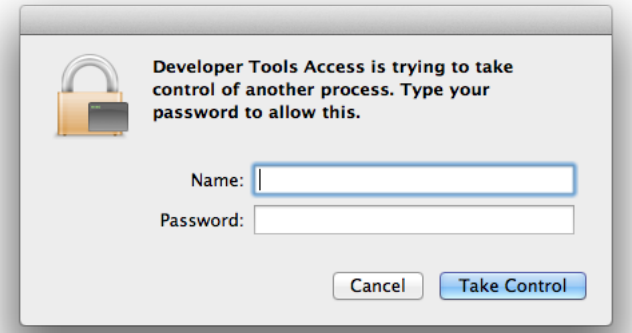


Figure 5.2: Authorization prompt to use the Developer Tools

Hiatus with Calabash

The Calabash integration process with Jenkins suffered a sudden stop due to administrative decisions. Another affiliate that also had automated acceptance tests was using Frank for quite some time, an framework in everything similar to Calabash as described in Chapter 3.

In spite of the reasons presented in the Related Work to use Calabash instead of Frank, the company wanted by all means to use Frank because many tests and steps definitions could be reused.

This decision resulted in a hiatus with Calabash to give preference to Frank which led to a new cycle of research to find if Frank could substitute Calabash. At this juncture, Frank was integrated with the project—a process that by itself revealed to be more painful than with Calabash—and some experiments followed. As the research previously conducted during the literature review allowed to conclude, Frank was inferior when compared to Calabash. These new tests guided to two conclusions: Frank had poor gestures support and it did not offer support for testing with devices, a great objective of the continuous delivery solution.

These findings deterred the administration from insisting on using Frank which led to breaking the hiatus and returning to Calabash.

Concurrent tests

The current solution does not count with parallel tests since only one iPad is being used for now. However, some research was conducted to investigate the possibility; additionally, a solution was implemented with the capacity to run Calabash on multiple devices at the same time.

Calabash has two ways of launching applications: `SimLauncher`[14] and Apple's Instrument tool. For iOS 7—the target version for testing at Blip—only the Instruments launch method is supported. Not using it to launch automatically the applications will result in touch events not working.

The bottleneck to run concurrent tests is the Instruments because there can only be one process running at any one time. Concurrent testing is possible when launching without Instruments which is not viable when testing on iOS 7.

A solution involving virtual machines was studied and even tested. Through virtualization, it is easy to have a running system with a hypervisor[15] with the minimum required configurations and dependencies to run Calabash. With VMware Fusion[16], a virtual machine was created with OS X Mavericks. Afterwards, all the required software to run Calabash was installed, as already mentioned in the previous sections.

With Calabash correctly working on the virtual machine (VM), it was a matter of creating a communication channel between the VM and Jenkins. For this purpose, nothing better than taking advantage of the Jenkins master/slave mode. With such approach, one can run multiple slaves on the same machine. Each VM runs Calabash on a different device.

This solutions allows to test on multiple devices at the same time and it can be improved, as it was slightly tried, to scale dynamically according to the connected devices. Algorithm 5.1 illustrates this dynamic approach which in fact is a evolution of the Algorithm 4.1. Two Jenkins jobs are used: a master job that orchestrates the slaves management as well the virtual machines allocation to the devices and a job for the automated acceptance tests that is executed in each slave. These two jobs have the same workflow as previously shown in the beginning of this section but now they are partitioned: the first three steps are executed on the master job, while the last two get executed on each slave.

[14] https://github.com/moredip/Sim-Launcher

[15] A hypervisor or virtual machine monitor (VMM) is a piece of computer software, firmware or hardware that creates and runs virtual machines.

[16] http://www.vmware.com/products/fusion/

Algorithm 5.1: Automated acceptance tests on multiples devices

Require: *sourceCode*

```
1: build ← Build(sourceCode)
2: ipa ← PackageApplication(build)
3: devices ← GetDevices()
4: addresses ← LoadAddresses()
5: if Empty(devices) then Exit(1)                    ▷ Abort execution
6: end if
7: SetupVM(ipa, features)
8: CloneVM(Count(devices))
9: slaves ← LaunchVMs()
10: for i ← 0, Count(devices) do
11:     deviceType ← Type(devices[i])
12:     deviceId ← Id(devices[i])
13:     deviceAddress ← addresses[deviceId]
14:     CalabashJob(slaves[i], deviceType, deviceId, deviceAddress)
15: end for
```

The algorithm illustrates the main workflow of this approach. Initially, there is only one VM. This virtual machine is the one who gets the `.ipa` file and the Cucumber features. Additional necessary resources are also added to this VM. This way, instead of having, lets say, five machines for five devices and setting up each one, all the process is done on a single machine and then that same machine gets duplicated to match the number of devices to be tested. If hardware resources are a concern, instead of having a VM for each device, a limit of the number of machines can be configured and the devices are partitioned. For instance, if the limit is 2 virtual machines and there is 4 devices, each VM will test two devices. This load balancing method minimizes the time it takes to execute the tests on all the devices.

5.3 Deployment pipeline improvement

To improve the deployment pipeline, a lot of experimentation allied with trial and error was required. Jenkins is very extensible but making it do the right thing in the right way is not always a easy job.

The first great goal was to modularize the pipeline. Such task involved a lot of researching. Jenkins is somewhat limited regarding orchestration of jobs. A lot of plugins were used and reused, some of them more than once. Build Flow, after so many experiments, ended up being the best candidate.

The division of the jobs labor was the second goal. The initial division is not comparable with the current one. This process of modularization has undergone various changes over time. Often, something that made sense in one job had to be reallocated to another. In other cases, due to technical limitations, it was not possible to add features that would improve the overall process; rather, far-fetched solutions were put into place.

The creation of the `Pipeline` and `Setup` jobs allowed to purge repeated logic on the `Build` and `Unit tests` jobs. Instead of pulling twice the code from the VCS and installing the dependencies, that process was moved these new jobs and from this point on a single workspace was used. This approach, more than anything, ensured that the same codebase was used on the entire pipeline. At this point, the deployment pipeline had four jobs: *Pipeline*, *Setup*, *Unit tests* and *Build*.

The job with Calabash was added to the pipeline and it was getting called after every successful build. However, soon, the decision to separate the automated acceptance tests from the pipeline was made because of the time it was requiring to terminate the whole pipeline. By way of example, currently, the complete test suite that contains 29 Cucumber features takes an average of 30 minutes to test.

A lot of experimentation was done on the parallelization of the jobs to be able to run multiple instances of the same job at the same time on different Jenkins slaves. This research was important to be able to run Calabash concurrently on multiple devices as previously explained. Such concurrency was successfully achieved through the Build Flow plugin.

The build process was optimized as well. Xcode intensively cache temporary information on the file system. As result of this caching, the speed of building and launching the application in debugger may significantly depend on the *speed* of the hard drive[17]. With SSD, a typical iOS-project with precompiled information will launch in 5–10 seconds. However, the cached files were not being reused which was slowing down the build process.

Unit tests were another subject of optimization. The use of xctool and its ability to run tests in parallel have accelerated altogether the process. Moreover, the functionality of the reports helped to generate more complete reports and metrics.

Lastly, the internal release of the `.ipa` file was moved to a completely separate job. As a result, the *Build* job longer has been broken due to errors not related to the build process itself.

Overall, the modularization, refactoring and optimization of the deployment pipeline resulted in a feature-rich pipeline, more automated and with faster feedback. The average execution time of the pipeline went from 20 minutes to 15, which in a commit basis environment is a big improvement that results in less waiting times and, consequently, shorter build queues.

[17] http://blog.shpakovski.com/2014/02/how-to-reduce-xcode-and-appcode.html

Chapter 6

Case Studies

The Merriam-Webster dictionary[1] defines *case study* as "a published report about a person, group, or situation that has been studied over time" and also as "a situation in real life that can be looked at or studied to learn about something".

This chapter describes three case studies developed within this dissertation and gives a broad explanation on how to reproduce the solutions by anyone.

Section 6.1 deeply describes the essential tools required to obtain a solution like the one presented in this dissertation and Section 6.2 presents three variations of the solution related with how the automated acceptance tests are executed.

6.1 Fundamental Recipe

The whole solution revolves mainly around two concepts: *automation* and *feedback*. These two concepts are the very fundamental ingredients to obtain a minimal working solution.

For the first one, continuous integration is the key. As a matter of fact, a continuous integration server is the very central piece of the solution as it is the foundation of the solution, and so it assumes an important role as long-term tool. Therefore, the decision on the tool to be used must not be taken lightly.

With so many CI servers to choose from, it can be difficult to decide which one is the right for the job. The first decision one have to make is whether to use a hosted SaaS solution or a self-hosted maintained server. If security is a concern then a locally installed server is the better choice. SaaS solutions are generally easier and simpler to setup but they also only offer a predefined set of features. If customizability is a necessity, then a self-hosted environment is more suitable. To use whether open-source software (OSS) or proprietary software is another topic to give some thought. Generally, OSS software gets

[1] http://www.merriam-webster.com/dictionary/case%20study

more support from the community. Lastly, there are CI tools that focus on CI but not so much in continuous delivery.

With an open-source self-hosted solution in mind, Jenkins was the better solution. It has a vast community of supporters with hundreds of plugins[2]. Because of its self-hosted nature, it offers a high level of customization and security.

For the second main concept, feedback, a modular deployment pipeline is the key. As previously shown in the Chapters 2 and 4, a pipeline is an "automated manifestation of the process for getting software from version control into the hands of the users" [HF10]. Every change to the software goes through a complex process on its way to being released. This sequence of test stages, each evaluating the build from a different perspective, allows to measure the build viability to become a production release. As the build passes each step, confidence in it increases. Any build that fails a stage in the process is not promoted the next stage.

Every single stage of the pipeline must return in some way feedback to the teams, as shown in the sequence diagram in Figure 2.2. For this purpose, proper setup and tuning of the CI server is required. Figure 2.2 also reveals the required steps to have a deployment pipeline: *version control*, *build*, *unit tests*, *automated acceptance tests*, *manual tests* and *release*. Manual tests and release do not make part of the CI server but they are fundamental for a continuous delivery solution. As for the rest of the steps, they should be described as jobs in the CI server and the more refined they are, the more detailed feedback will be possible to collect from them.

Making these steps independent and self-contained makes it possible to provide fast feedback. In addition, by running the fastest steps early in the pipeline, allows the delivery team to quickly understand that the build is broken.

An important ingredient in this pipeline is *refinement*. Each job must be properly setup to provide the most complete feedback possible. Having a job dedicated to the job orchestration makes the pipeline much easier to understand and reconfigure. Changing the steps order, removing unnecessary steps or adding new ones becomes very simple only requiring the update of a unique script which describes the overall flow of the pipeline.

The implemented solution contains the job `Setup` that may be optional depending of the necessities of the project where the pipeline is being applied. `Setup` verifies and manages the dependencies of the project. It can also prepare the environment and the workspace for the further steps of the pipeline. This job represents a clean way to setup the the environment and having it separated from the rest of the flow keeps the pipeline modular, an important characteristic of the developed solution. Relatively to the remaining steps of the pipeline, all of them require special cares.

The `Unit tests` job, as already described, executes the unit tests and generates code coverage metrics. Apple's Xcode provides a command line tool to build and test projects

[2] https://updates.jenkins-ci.org/download/plugins/

called **xcodebuild**. However, to execute the unit tests, this job is using **xctool**, a replacement for xcodebuild created by Facebook that makes it easier to test iOS (and Mac) products. It was created especially for continuous integration and adds a few extra features compared to the Apple's tool. It has a more structured output and it even generates it as JSON, which is very useful for using with another plugins to generate reports. The other big advantage is the ability to run the test bundles in parallel, speeding up the tests significantly.

The project is also configured to generate test coverage files with the GCOV code coverage tool. `gcov` uses two files for profiling: `.gcno` and `.gcda`. The names of these files are derived from the original *object* file by substituting the file suffix with either `.gcno` or `.gcda`[3]. The files contain coverage and profile data stored in a platform-independent format.

Xcode generates `.gcno` files when compiling the application and `.gcda` data files when executing the application. The `.gcno` notes file contains information to reconstruct the basic block graphs and assign source line numbers to blocks. The `.gcda` count data file is created for each object file compiled and it contains arc transition counts, value profile counts and some summary information.

After the execution of all unit tests, `gcovr` is used to generate a XML report in Cobertura format. Cobertura[4] is a tool that calculates the percentage of code accessed by tests. By generating a report in such format, it is then possible to use the Cobertura Plugin[5] which generates a human-readable report of coverage.

At the *build* stage, the `Build` job simply compiles the project. If the build process terminates successfully, the job also generates an `.ipa` to be uploaded to TestFlight later on the `Deploy` job.

The main pipeline ends with the `Deploy` job which only uploads the previously generated `.ipa` file to the TestFlight servers for internal release.

The automated acceptance tests stage is not included in the deployment pipeline due to its lag when compared to the rest of the pipeline. Nonetheless, such delay on the automated acceptance tests may vary from project to project. The quantity of tests and the flow of the application under test (e.g., the time it takes to go from one screen to another or the time it takes the application to communicate with the server) are examples of factors that greatly influence the overall duration of the tests. On circumstances where automated acceptance tests can be included in the deployment pipeline, and due to the use of a master job to orchestrate the pipeline flow, the only necessary change is to add one instruction to call the job with the automated acceptance tests to the build flow. Listing 6.1 illustrates the deployment pipeline flow with automated acceptance tests. When compared to the Listing 4.1, only one instruction was added, and yet the pipeline this way would start to

[3] https://gcc.gnu.org/onlinedocs/gcc/Gcov-Data-Files.html
[4] http://cobertura.github.io/cobertura/
[5] https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin

run the automated acceptance tests after a successful build. Moreover, the deploy phase would only be triggered if the automated acceptance tests terminated without errors.

```
1 ...
2 job_setup = build("Setup", PROJECT_DIR: workspace)
3 job_unit_tests = build("UnitTests", PROJECT_DIR: workspace)
4 job_build = build("Build", PROJECT_DIR: workspace)
5 ...
6 job_acceptance_tests = build("AutomatedAcceptanceTests", PROJECT_DIR: workspace)
7 job_deploy = build("Deploy", PROJECT_DIR: workspace, IPA: ipa)
```

Listing 6.1: Deployment pipeline with automated acceptance tests

Whether driving the functional tests on the simulator or on the devices, the overall workflow of the Jenkins job is very similar, as already mentioned in the Section 4.3. First, the application under test must be built with the Calabash framework. Then, if one means to test on a device, the application must be converted into an `.ipa` and sent to the device. When testing with devices, three informations are required: its UDID, its Wi-Fi address and whether it is an iPhone or an iPad (some tests may be meant to run only for one type of device).

6.2 Variations

The deployment pipeline with automated acceptance tests falls in three variations. These variations are put into practice if the form of Jenkins jobs. They may be included as stage of the deployment pipeline or they can be executed independently.

Automated acceptance tests on simulator only

With this variation, the automated acceptance tests are only executed on the iOS simulator, whether on the iPhone or the iPad mode. Calabash performs much faster on the simulator than on devices and so this is the advantage of this variation. However, as already presented in Chapter 1, testing only on the simulator is not a good approach. On one hand, some features are not reproducible on the simulator and, on the other hand, the intrinsic behavior of the simulator is very different when compared to the physical devices, leading to possible unexpected results.

This variation is not only the fastest but also the most suitable when the intended is to include the automated acceptance tests in the deployment pipeline. However, it is more error-prone than the next variation.

Automated acceptance tests on devices only

This approach aims in testing only on devices, whether in parallel or not. Because Calabash communicates with the application under test through an HTTP server, testing with physical devices results in a slower exchange of commands. The application itself is faster on the simulator when communicating with online services since the simulator uses the best available channel of network communication, which allows to use an Ethernet connection; in contrast, the best channel on a device is the Wi-Fi connection which by nature is slower. When executed separately from the pipeline, they are not triggered on a commit basis; rather, they are scheduled to run at specific times.

Mixed solution

This variation is self-explanatory. The intention here is to find a middle ground between the previous solutions. Testing on simulator is faster but more error prone, while testing on devices gives more realistic results which, consequently, results in greater levels of confidence on the application. The current solution implemented at Blip follows this approach.

The main idea is simple: test on the simulator regularly with small time intervals and test on devices at the midnight of every day. This approach translates itself into two different Jenkins jobs where none makes part of the main pipeline. Again, the decision of integrating these jobs in the pipeline or not depends on the time it takes to complete all the tests.

Chapter 7

Critical Discussion

This chapter gives a thorough analysis of the work done withing this dissertation.

Section 7.1 describes briefly the change that occurred on the scope of this dissertation and its implications. Section 7.2 gives a detailed insight of the decisions made and problems encountered on the development of the overall continuous delivery solution, while Section 7.3 focus specifically on the automated acceptance tests and the path taken to integrate them on the solution. Lastly, Section 7.4 discriminates the answers to the research questions presented in the Chapter 1.

7.1 Overview

This dissertation main goal on the beginning was the definition of a test automation system for mobile applications. The goals defined by Blip were to research, develop and document a system with automated tests for different devices. The outcome expected of the project was a full automated system of automated acceptance tests that could be used to run tests across multiple devices "with no need to have them physically in the same space", as defined on the dissertation specification.

Soon, due to the intrinsic relation with the overall process of software development, the scope of the dissertation extended covering the whole pipeline of a continuous integration solution. This change on the scope not only enlarged the task list as it increased the overall difficulty of the dissertation as well. The goal went from managing one step of the pipeline to managing the whole pipeline.

7.2 Deployment Pipeline

At the beginning of the dissertation, Blip had a basic CI solution featuring a simple pipeline with two steps: one for building and another for driving unit tests. To improve this

pipeline, and since such task was a recent goal, some research was conducted. Most of the research was done in parallel with a trial and error method, a method that was very common throughout the dissertation.

Following predominantly this method was not sign of carelessness or lack of a methodology. Instead, it was sign of the complexity to describe apparently simple behaviors and workflows in the form of jobs, the main logic units of Jenkins, the CI tool used that served as the foundation of everything else.

Jenkins was already being used at Blip and, in spite of the given freedom to change to another tool, it remained as the chosen one. Not only it is open-source with a big supporting community, as it strikes over other CI solutions due to its customizability as well. Jenkins counts with a big collection of plugins which makes it very extensible and almost ready for anything. The ability to write plugins is a great advantage but, better yet, one can augment the system functionality without even going through the trouble of writing a script by just taking advantage of the built-in support for Groovy script that permits to access all the internals of Jenkins in a very easy way. The downside of Jenkins is probably its interface and the poor user experience it provides.

To optimize and refactor the pipeline, a thorough knowledge of the current proceedings was needed. The deployment scripts were studied and the inner workings of the Xcode understood. It was specially important, during the dissertation, to comprehend how the compiler and linker worked in order to optimize the building process as well as possible. For instance, understanding how Xcode cached the builds was important to minimize the build times on Jenkins.

Acclimatisation with scripting languages was another essential requisite along the deployment pipeline improvement. Automation speaks scripting languages, not GUIs. If Jenkins is the foundation of the solution, Bash scripts are the beams of it. Scripting languages are written for a run-time environment that can interpret (rather than compile) and automate the execution of tasks. Bash, the shell of OS X and other Unix-like operating systems, is very powerful.

In order to obtain a more robust deployment pipeline, the decision of turning the jobs into smaller units was made. Eliminating the repeated logic from the initial jobs by separating and isolating the code checkout from the building and unit testing stages required some discipline and refactoring. The orchestration of the jobs represented a over-complicated problem. Jenkins should provide more features regarding the chaining of jobs and offer native ways to communicate and exchange artifacts between them.

Regarding unit tests, xctool was the chosen tool rather than the Xcode command line tool. In the end, the switch was worthwhile. By enabling parallel tests, the average duration of the tests was reduced. The tool's reporters with better outputs were also a great advantage, mainly for using with third-party plugins. For the code coverage reports, gcovr was the tool used. Research led to the conclusion that it was the best candidate for the job. However, this tool proved to have some bugs: it did not work properly when specifying

an absolute path of the code coverage files and the HTML reporter was broken failing most of the times without a clear error message.

Xcode and its provisioning profiles revealed to be hard to troubleshoot when something went wrong. Building iOS applications with the wrong provision profiles due to misconfigurations can result in problems hard to trace. The archiving and code signing step most of the times does not fail; rather, the upload to TestFlight would fail with no apparent reason or the deployment to the devices for the automated acceptance tests would not succeed. Even worse, some times the `.ipa` file was successfully deployed on the device but then Calabash could not launch the application.

The deployment pipeline developed within this dissertation is superior in all respects when compared to the starting point pipeline. Not only it is more abundant in features, as it is quicker as well. The current pipeline takes in average 15 minutes to complete, five minutes faster than its early version which represent a great improvement when considering that it is triggered on a commit basis. It makes a big difference. The average duration decreased but the completeness greatly increased. It runs entirely on the same codebase from the beginning to the end, runs unit tests in parallel, calculates code coverage metrics, generates human-readable reports, builds take less time and are less error-prone and problems related to the internal release process no longer breaks the `Build` job itself.

Overall, the pipeline is more readable through its new view. *Automation* and *feedback* were the keywords since the beginning to build a robust solution that could indeed improve the **quality** of the produced software. The developed solution succeed in such goal. It is taking less time to validate the quality of the product, there are more indicators of confidence, including code coverage metrics that break the build when the coverage is below defined minimum values. The modularity of the pipeline provides fast feedback.

7.3 Automated Acceptance Tests

Automated acceptance tests where the initially goal of the dissertation and they were meant to be integrated in the deployment pipeline. This kind of testing on the mobile environment is yet a recent paradigm.

Calabash proved to be the best choice, even when its integration suffered a stop and Frank was explored. The integration with the Blip iOS project was somewhat laborious. First, and as previously mentioned, an alternative method to the Calabash Xcode target was investigated to facilitate the project maintenance. This approach required some tweaks on the project in order to work. Second, the first time Calabash was integrated with the project, the distribution scheme was used, the one that is used by the teams to distribute internally the application and to test it on the production environment. It took some time to figure out a serious problem that was happening on the application. The application started to behave very slowly without apparent reason. It took some time to

understand that it was the Calabash HTTP that come into conflict with the normal behavior of the application. Knowing the problem, its resolution was not difficult. A new scheme was created, identical to the distribution one, to link the application with Calabash. This new scheme is used only for the automated acceptance tests.

Calabash has two limitations that leave room for future improvements. First, the instrumentation on devices is slower than with the iOS simulator. Such behavior is not favorable and the cause is not obvious. It can be due to the protocol communication implemented by Calabash or the Apple's UIAutomation API. iOS 7 is relatively recent[1] and brought many changes that forced Calabash to adapt its inner workings, an adaptation that continues at this moment. The second limitation is not entirely a problem of Calabash but of Instruments, that it not supports multiple instances, resulting in the impossibility of having native parallel tests.

Overall, Calabash corresponded with the expectations created during the research of frameworks for automated acceptance tests. Its support for testing web views was fundamental as the application under test had some views of this kind. Frank, on the other hand, could not do such testing since it not supports web views. The late administrative choice to use Frank was an unexpected problem. At that moment, all the research conducted and knowledge obtained was done on Calabash. Fortunately, before long, with some additional research and some practical experiments, it was possible to persuade that Calabash was the best choice. The only downside of Calabash, when compared with Frank, is the lack of an inspector. Symbiote, Frank's inspector, is a web application lets one to inspect the current state of the application being instrumented and allows to query UI elements and test the selectors. Such tools provides a tree describing the current state of the view hierarchy of the running application. This is indeed a good feature that accelerates the development of the automated acceptance tests.

7.4 Research Questions

In Chapter 1, some questions were presented. The answers to those questions follows.

What is the proper approach to take to have an automated pipeline?

This question stood up initially as a mystery. The big conclusion is that there is not a *right* implementation or a perfect solution. To implement continuous integration is to create a paradigm shift in a team [HF10]. Without CI, an application is broken until one prove otherwise. With CI, the default state of the application is working, albeit with a level of confidence that depends upon the extent of the automated test coverage. CI creates a tight feedback loop which allows one to find problems as soon as they are introduced.

[1] iOS 7 was publicly released on September 18, 2013.

In other words, the company culture and the team workflow where the continuous integration is implemented affects the whole process. That is to say that depends from case to case. For instance, the pipeline implemented within this dissertation did not include the step of automated acceptance tests due to time constraints.

How to improve the level of confidence on the software under test?

A continuous integration solution is only worth if it can give a high level of confidence to the team. The idea is to have a level so high that if the pipeline is not broken it means the application is fully working and the team has confidence on it.

Within this dissertation, to have automated acceptance tests was a great goal. It was a inexistent step before the dissertation and its integration was desirable. It assumes an important role on the CI solution as it represents a high level of confidence. Code coverage metrics on unit tests were another important feature to provide not only confidence but feedback as well.

How to connect multiple mobile devices in the network?

This question arose following the idea of having the devices under test connected via a wireless channel. Soon, this approach proved to be not only unreliable but impracticable as well. Technical limitations did prevent such kind of implementation.

How to collect back the testing results from the devices?

Similarly to the previous question, this question arose before precisely knowing what framework or tool that would be used for conducting the automated acceptance tests. It turned out that Calabash itself could handle the results retrieving.

Logical tests are covered but how to enforce usability tests that only real people can lead?

Manual tests are part of the deployment pipeline, or at least they should be, as shown in Figure 2.2. Theses tests are done by people as no machine can do them. This requires discipline and it is responsibility of the team to decide the course of action.

Chapter 8

Conclusions

This last chapter closes this dissertation with a set of conclusions regarding all the work involved with the research conducted and the solution implemented.

Section 8.1 presents by way of reflection the final thoughts is respect of this dissertation. To conclude, Section 8.2 points out by way of future work the next possible approaches to further improve the solution developed.

8.1 Reflection

With the non-stop proliferation of mobile devices, ever more is necessary to thoroughly test the mobile applications under development. Such tests greatly improves companies confidence degree, mainly if done on physical devices—if the application works on a dozen of mobile devices in an environment close to the production, it will be less error-prone and the overall users satisfaction will be bigger.

This dissertation encompassed initially the definition and implementation of an automated system for testing mobile applications. Its scope quickly escalated to a broader solution which became the improvement of a continuous integration solution, already in place at Blip but still on its early steps. Moreover, this CI solution should be optimized in order to be converted into a continuous delivery solution. All these requisites were synonym of a desired formula whose main objective was to improve the software quality in an automated fashion.

Regarding acceptance tests, manual testing was out of question because of its obvious disadvantages. Mobile test automation arose as the answer. Cloud-based testing services do exist but cloud computing is yet a risk that most companies do not want to take. Security vulnerabilities, privacy concerns, possible outages and downtimes prevent companies from adopting these modern architectures. Moreover, self-hosted solutions are far more attractive because offer ways of extensibility that cloud services do not.

A methodical process that lives along with continuous integration practices and builds the application under test and then tests it on several real mobile devices can greatly decrease the delivery cycle.

The developed solution could not be added to the main deployment pipeline due to time constraints. Yet, it runs in defined periods of time providing feedback to the team of the current state of the application.

Regarding the deployment pipeline, its optimization and refactoring resulted in visible results. It became quicker, more responsive, more legible and, above all, less error-prone.

Automation and feedback were the most important concepts to have in mind during the implementation of the pipeline. But the solution takes only into account the technical part of the deployment pipeline. On the subject of feedback, it is no good unless it is acted upon. This requires discipline and planning. It is the responsibility of the whole team to stop what they are doing and decide on a course of action whenever something requires attention.

8.2 Future Work

The deployment pipeline went from an average duration of 20 minutes to 15 but there is still room for improvement regarding the total time of execution. Removing the internal release of the pipeline is a possible approach. That is not to say removing entirely but making the step asynchronous; in other words, the pipeline would end after the termination of the *Build* job, yet a job on the background would upload the software under test to the web. This upload is currently taking in average 7 minutes which is almost half of the total time. Since this step is not a showstopper, in the first place, it should not break the pipeline just because the upload failed—such behavior should not break the build—, and, second, it should not slow down the overall pipeline, delaying all the build queue.

Regarding the automated acceptance tests, a lot of research can be conducted to improve the current implementation. Test suite minimization, selection and prioritization represent topics of high interest that can improve immensely the automated system of regression tests presented in this dissertation.

The most straightforward approach of regression testing is to simply execute all the existing test cases in the test suite. This is called a *retest-all* approach [YH12]. This approach is the one being used with Calabash and the written automated acceptance tests within this dissertation. As previously mentioned, the current test suite takes more than 30 minutes to complete and as test suites tend to grow over time, it may be prohibitively expensive to execute the entire test suite [YH12, CA13]. This limitation leads to considerate techniques that seek to reduce the effort required for regression testing in various ways. The three major branches regarding these techniques are *test suite minimization*, *test case selection* and *test case prioritization*.

"Test suite minimization is a process that seeks to identify and then eliminate the obsolete or redundant test cases from the test suite. Test case selection deals with the problem of selecting a subset of test cases that will be used to test the changed parts of the software. Finally, test case prioritization concerns the identification of the 'ideal' ordering of test cases that maximize desirable properties, such as early fault detection. Existing empirical studies show that the application of these techniques can be cost-effective." [YH12]

[CA13] proposes an approach, RZoltar, that addresses this problem of an ever-growing test suite with a solution of minimization. They show that RZOLTAR efficiently (0.95 seconds on average) finds a collection of test suites that significantly reduce the size (64.88% on average) maintaining the same fault detection (as the initial test suite), while the state-of-art greedy approach [TG05] needs 11.23 seconds on average to find just one solution.

Test prioritization is another potential candidate to improve the developed solution. Prioritized regression test suites aim at detecting failures as soon as possible in order to reduce testing effort. Automatic fault localization techniques use the information obtained during regression testing to produce a ranking of source code statements likely to be the root cause of the observed failures. This ranking is used to minimize the diagnostic work the developer has to perform when inspecting the program to find the faults. Raptor, is a test prioritization algorithm for fault localization, based on reducing the similarity between statement execution patterns as the testing progresses, with benchmarks proving that it is the best technique under realistic conditions [GSAGV11].

These two techniques allied with the solution presented on this dissertation can vastly improve the regression testing by eliminating redundant tests cases and producing a ranking of the most likely causes of failures.

Test generation is another field of study that can be integrated within the automated system presented on this document. [CAFD13] presents an approach to produce new test cases automatically through an technique that borrows ideas from probability theory.

References

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the Clouds: A Berkeley View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[BC13] Shilpa Bahl and M M Chaturvedi. Literature Review of Mobile Applications Testing on Cloud from Information Security Perspective. *International Journal of Computer Applications*, 79, 2013.

[BXX07] Jiang Bo, Long Xiang, and Gao Xiaopeng. MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, Washington, DC, USA, 2007. IEEE Computer Society.

[CA13] Jose Campos and Rui Abreu. Leveraging a Constraint Solver for Minimizing Test Suites. In *2013 13th International Conference on Quality Software (QSIC)*, pages 253–259. IEEE, 2013.

[CAFD13] José Campos, Rui Abreu, Gordon Fraser, and Marcelo D'Amorim. Entropy-based test generation for improved fault localization. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 257–267. IEEE, 2013.

[Cis13] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2012–2017. Technical report, Cisco, 2013.

[CR12] Inderveer Chana and Ajay Rana. Empirical evaluation of cloud-based testing techniques: a systematic review. *ACM SIGSOFT Software Engineering Notes*, 37(3):1–9, 2012.

[DS14] Stephen M Dye and Karen Scarfone. A standard for developing secure mobile applications. *Computer Standards & Interfaces*, 36(3):524–530, March 2014.

[FF06] Martin Fowler and Matthew Foemmel. Continuous integration. *ThoughtWorks) http://www. thoughtworks. com/Continuous Integration. pdf*, 2006.

[GSAGV11] A Gonzalez-Sanchez, R Abreu, H.-G. Gross, and A J C Van Gemund. Prioritizing tests for fault localization through ambiguity group reduction, 2011.

[HF10] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[HLHG13] Shuai Hao, Ding Li, William G J Halfond, and Ramesh Govindan. SIF: A Selective Instrumentation Framework for Mobile Applications. 2013.

[HRN06] Jez Humble, Chris Read, and Dan North. The deployment production line. In *Agile Conference, 2006*, pages 6–pp. IEEE, 2006.

[IAS12] Koray Inçki, Ismail Ari, and Hasan Sozer. A survey of software testing in the cloud. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*, pages 18–23. IEEE, 2012.

[JMMN+99] Matt Jones, Gary Marsden, Norliza Mohd-Nasir, Kevin Boone, and George Buchanan. Improving Web interaction on small displays. *Computer Networks*, 31(11–16):1129–1137, May 1999.

[Jor13] Paul C Jorgensen. *Software testing: a craftsman's approach*. CRC press, 2013.

[LSL07] Andres Löh, Wouter Swierstra, and Daan Leijen. A principled approach to version control. Citeseer, 2007.

[MW13] Mary Meeker and Liang Wu. 2013 Internet Trends. Technical report, Kleiner Perkins Caufield Byers, 2013.

[Roc10] Francisco Emanuel Liberal Rocha. *Privacy in Cloud Computing*. Master thesis, Faculty of Science of the University of Lisbon, 2010.

[Sea00] AndrewJacko Sears Julie A. Understanding the Relation Between Network Quality of Service and the Usability of Distributed Multimedia Documents. *Human-Computer Interaction*, 15(1):43–68, March 2000.

[SS13] Graeme Sewell and Immanuel Simonsen. The Webcertain Global Mobile Report 2013. Technical report, Webcertain, 2013.

[SV13] Oleksii Starov and Sergiy Vilkomir. Integrated TaaS Platform for Mobile Development: Architecture Solutions. In *Proceedings of the Eighth International Workshop on Automation of Software Test (AST 2013), San Francisco, USA, in conjunction with the 35th International Conference on Software Engineering (ICSE 2013)*, 2013.

[TG05] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 35–42. ACM, 2005.

[TKHD06] D Talby, A Keren, O Hazzan, and Y Dubinsky. Agile software testing in a large-scale project. *IEEE SOFTWARE*, 23(4):30+, 2006.

[TTK09] Motoharu Takao, Susumu Takahashi, and Masayoshi Kitamura. Addictive personality and problematic mobile phone use. *CyberPsychology & Behavior*, 12(5):501–507, 2009.

[Vil12] Sergiy Vilkomir. Cloud Testing: A State-of-the-Art Review. *Information & Security: An International Journal*, 28(2):213–222, 2012.

[YH12] S Yoo and M Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, March 2012.

[YTC+10] Lian Yu, Wei-Tek Tsai, Xiangji Chen, Linqing Liu, Yan Zhao, Liangjie Tang, and Wei Zhao. Testing as a Service over Cloud. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pages 181–188. Ieee, 2010.

[ZA05] D Zhang and B Adipat. Challenges, methodologies, and issues in the usability testing of mobile applications. *International Journal of Human-Computer Interaction*, 18(3):293–308, 2005.